Radio Shack®

TRS-XENIX™ Operating System

MBASIC Interpreter

(Multi-user)

CUSTOM MANUFACTURED IN U.S.A. BY RADIO SHACK, A DIVISION OF TANDY CORPORATION

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

Addendum

Changes to the

TRS-XENIX MBASIC Manual

The following changes should be made to your TRS-XENIX MBASIC Interpreter Reference Manual:

page	change
74	LOF returns the number of bytes in a file, not the end-of-file record number as stated.
87	CLEAR -- The first example should read: CLEAR ,61000, 2000 clears all variables and closes all files; allocates 61000 bytes for program and variable space, and 2000 bytes for stack space.
102	DIM -- if an array type is not specified, the array is classified as double-precision, not single-precision.
134	KILL -- You may kill a file that is open. This follows TRS-XENIX conventions.

164 OPEN -- The option RANDOM for mode does not exist. If the mode is omitted, RANDOM is automatically selected.

Also, the record length can be as large as 32715 bytes, not 32767 as stated.

Note: The maximum number of files that can be opened at one time is 17.

194 SAVE -- notes on the P option:

The P option of SAVE tells MBASIC to save the program as a protected file. A protected file is a binary encrypted BASIC program that has been scrambled by a special algorithm.

MBASIC allows only certain statements to be executed while a protected file is in memory. They are:

RANDOMIZE	CONT	DATE$	FILES
KILL	LOAD	NAME	NEW
RUN	SYSTEM	TIME	

An error occurs if you try to use any other statement.

Also, the example at the bottom of the page should read:

```
SAVE "MATHPAK.TXT", A
```

saves the resident program in ASCII form, using the name MATHPAK.TXT.

Radio Shack
8759329

ADDENDUM

MBASIC Reference Manual

A new keyword has been added to TRS-XENIX BASIC. Please insert the attached page to add "ON BREAK" to your MBASIC Reference Manual.

Thank You!
Radio Shack
A Division of Tandy Corporation

ON BREAK GOSUB Statement

ON BREAK GOSUB line

Transfers control to line if <BREAK> is pressed.

Normally <BREAK> terminates the program execution. But ON BREAK lets you use the <BREAK> key as a switch to execute the specified subroutine.

Example

```
10 CLS
20 ON BREAK GOSUB 70
30 PRINT@ I, "*"
35 I = I + 1
40 IF I < 0 OR I > 1919 THEN 130
50 GOTO 30
60 RETURN
70 ON BREAK GOSUB 20
80 PRINT@ I, "."
90 I = I - 1
100 IF I < 0 OR I > 1919 THEN 130
110 GOTO 80
120 RETURN
130 END
```

The first part of the program prints asterisks (*) starting at Position 0. When you press the <BREAK> key, control passes to subroutine beginning at Line 70. This subroutine prints periods (.). When you press <BREAK> again, control goes back to subroutine beginning at Line 20. When the cursor position is less than 0 or more than 1919, execution ends.

IMPORTANT

Installing MBASIC On Your Hard Disk

Please read all these directions before installing MBASIC on your hard disk. This lets you become familiar with the entire procedure and help make the installation smoother. During the installation be sure to watch the screen and answer all the prompts.

To begin installing the package:

1. Turn on all peripherals and then turn on the computer.
2. Log in as root (to install MBASIC, you must be logged in as root).
3. At the root prompt (#) type:

   install [ENTER]

   The screen shows Installation Menu. It also shows the prompts
   1) to install or q) to quit.
4. Enter [1] to install MBASIC.
5. At the prompt:

   Insert diskette in Drive 0 and press <ENTER>

   insert your TRS-XENIX MBASIC diskette in Drive 0 and press [ENTER].

The next screen shows the name of the package being installed, the version number, and the catalog number. At the bottom of the screen a welcoming message appears.

When TRS-XENIX finishes, the message:

Installation complete — Remove the diskette, then press <ENTER>

appears. Pressing [ENTER] returns the menu to the screen. You are then prompted to press:

1) to install another application, or
q) to quit.

Your installation of MBASIC is now complete. Press [Q] to quit and return to TRS-XENIX.

Note: You should never write protect your hard disk when running TRS-XENIX or MBASIC.

Radio Shack®
A DIVISION OF TANDY CORPORATION
FORT WORTH, TEXAS 76102

TRS-80®

TRS-XENIX MBASIC Interpreter

Reference Manual

Radio Shack®

Introduction

This is a reference manual for the TRS-XENIX MBASIC language. TRS-XENIX MBASIC is designed to run under the TRS-XENIX Disk Operating System.

TRS-XENIX MBASIC is an "interpreter." When you run a program, it executes each statement one at a time. This makes it quick and easy to use. It also allows you to take advantage many of TRS-8Ø features, such as:

- Faster running programs
- Larger programs

and TRS-XENIX features of:

- Multi-user
- Multi-tasking

About this Manual

This is a reference manual, not a tutorial. We assume you already know BASIC and are using this manual to quickly find the information you need.

Section I--Operations. This section shows how to load MBASIC. It also demonstrates how to write, run and save a MBASIC program on disk.

Section II--The MBASIC Language. This section includes a definition for each of MBASIC's keywords (statements and functions) in alphabetical order. In addition, it shows how to write a program to store data on disk.

If you have been using TRSDOS BASIC read Appendix D, "Differences Between TRSDOS BASIC and TRS-XENIX MBASIC" and Appendix C, "Transferring Files From TRSDOS BASIC to TRS-XENIX".

Terms and Notations

For clarity and brevity, we use some special notation and type styles in this manual.

CAPITALS	indicate material which must be entered exactly as it appears.
lowercase underlined	represent words, letters, characters or values you supply from a set of acceptable entries.
...(ellipsis)	indicates that preceding items may be repeated.
X'NNNN	indicates that NNNN is a hexadecimal number.
O'NNNN	indicates that NNNN is an octal number.
<keyname>	indicates one of the keys from your keyboard.
[ ](square brackets)	indicate that the enclosed entry is optional.
ß	indicates a blank space character (ASCII code 32). For example, in mbasic ßß progr there are two blank spaces between MBASIC and PROG.
line	a numeric expression that identifies a MBASIC program line. Each line has a number between Ø and 65529.
buffer	a number between 1 and 255. This refers to an area in memory that MBASIC uses to create and access a disk file. Once you use a buffer to create a file, you cannot use it to create or access any other file; you must first close the first file. You may only access a file with the buffer used to create it.

[parameters]	information you supply to specify how a command is to operate. Parameters enclosed in brackets are optional.
pathname	a sequence of directory names seperated by slashes (/) followed by a filename.
[arguments]	expressions you supply for a function to evaluate. Arguments enclosed in brackets are optional.
syntax	a command with its parameter(s), or a function with its argument(s). This shows the format to use for entering a keyword in a program line.
integer	any integer expression. It may consist of an integer, or several integers joined by operators. Integers are whole numbers between -32768 and 32767.
string	any string expression. It may consist of a string, or several strings joined by operators. A string is a sequence of characters which is to be taken verbatim.
number	any numeric expression. It may consist of a number, or several numbers joined by operators.
dummy number or dummy string	a number (or string) used in an expression to meet syntactic requirements, but whose value is insignificant.

This manual is organized as follows:

Section I. Operations

Chapter 1.	Sample Session
Chapter 2.	Command Mode Execution Mode
Chapter 3.	Line Edit Mode

Section II. The MBASIC Language

Chapter 4.	MBASIC Concepts
Chapter 5.	Disk Files
Chapter 6.	Introduction to MBASIC Statements and Functions
Chapter 7.	MBASIC Statements and Functions

Appendices

Appendix A	Error Messages
Appendix B	Reserved Words
Appendix C	Converting TRSDOS BASIC Programs to MBASIC
Appendix D	Differences Between TRSDOS BASIC and TRS-XENIX MBASIC
Appendix E	Machine Language Interface to TRS-XENIX MBASIC
Appendix F	Loading Assembly Language Files
Appendix G	Device Handling
Appendix H	Decimal Math Representation

Operations

MBASIC Reference Manual

Section I/ Operations

Chapter 1/ Sample Session

The easiest way to learn how MBASIC operates is to write and run a program. This chapter provides sample statements and instructions to help familiarize you with the way MBASIC works.

The main steps in running a program are:

A) Loading MBASIC
B) Typing the program
C) Editing the program
D) Running the program
E) Saving the program on disk
F) Loading the program back into memory

Loading MBASIC

After you power up your system and install the system diskette, the prompt "login:" appears.

To answer this prompt, type in your "login" name and press <ENTER>. (Refer to your XENIX operations manual for further information on the **mkuser** routine.)

You are now prompted to type your password.

Your password was also assigned during the **mkuser** routine. Type in your assigned password. The echo on the screen for your input is turned off for this prompt.

The words "Welcome to TRS-XENIX" now appear on your screen along with a dollar sign prompt: "$". This indicates that you are at the operating system level in your current directory.

To load MBASIC into the system, type:

mbasic<ENTER>

A paragraph with copyright information is displayed, followed by: OK

You may now begin using MBASIC.

Options for Loading MBASIC

When starting up MBASIC, you can also specify a set of options. They are:

mbasic [program] [-m memory size] [-e]

program	specifies a program to run immediately after MBASIC is started.
-m memory size	indicates how many bytes of memory are to be allocated for program, variable, and stack space. If omitted TRS-XENIX MBASIC attempts to allocate 32,767 bytes of memory.
-e	tells MBASIC to suppress echoing of user's input, the "OK" prompt, the "?" prompt associated with an INPUT statement and the sign on copyright information. This option is useful for MBASIC programs which are written to be TRS-XENIX filters.

To exit MBASIC use the SYSTEM command. Type:

system <ENTER>

to close all files and return to the operating system level in your current directory.(See chapter 7 for more information on SYSTEM.)

Examples

$ mbasic -m 45Ø56

initializes MBASIC; allocates 45Ø56 bytes of memory for program, variable, and stack space. MBASIC uses part of this memory allocation to initialize internal buffers; therefore only 43631 bytes are free for execution.

$ mbasic

initializes MBASIC in the command mode. After internal buffers there are 31Ø87 bytes of free memory.

```
$ mbasic -e
```

initializes MBASIC in the command mode but echoing of your input, the "Ok" prompt, the "?" prompt associated with an INPUT statement, and the sign on copyright information is suppressed.

Typing the Program

Let's write a small MBASIC program. Before pressing <ENTER> after each line, check the spelling. If you have made any mistakes while typing, use the <BACKSPACE> key to correct them.

```
1Ø A$="WILLIAM SHAKESPEARE WROTE " <ENTER>
15 B$="THE MERCHANT OF VENICE" <ENTER>
2Ø PRINT A$; B$ <ENTER>
```

Check your program again. If there is a mistake on a line, retype the entire line, including the line number, immediately after the current line.

For example, suppose you had typed:

```
15 B$="THE VERCHANT OF VENICE"
2Ø PRINT A$ B$
```

After line 2Ø, type

```
15 B$="THE MERCHANT OF VENICE" <ENTER>
RUN <ENTER>
```

Your screen should display:

```
WILLIAM SHAKESPEARE WROTE THE MERCHANT OF VENICE
```

MBASIC replaced line 15 in the original program with the most recent line 15.

MBASIC "reads" your program lines in numerical order. It doesn't matter if you entered line 15 after line 2Ø; it will still read and execute 15 before "looking" at 2Ø.

> Note: When you type in a program in lowercase, all the lowercase characters are converted to uppercase with the exception of the contents of string variables or constants and remarks (REM).

MBASIC has a powerful set of commands in the "line edit mode". This mode is discussed in Chapter 3.

Saving the Program on Disk

You can save any of your MBASIC programs on disk. To do this, you assign it a "filename".

For example, if you wanted to save the program we just wrote, you could assign it the filename "AUTHOR". Type the following command:

```
SAVE "AUTHOR" <ENTER>
```

It takes a few seconds for MBASIC to find a place to store our program. When this process is completed, it displays OK. The program is now saved on disk and stored in your current directory.

Filenames do not go through case conversion. They are saved exactly as typed. For example, the files "AUTHOR" and "author" will be saved exactly as typed; therefore they will be considered as two separate files.

> Important Note: A filename can also have an optional extension. A filename including an extension can have a maximum of fourteen characters. A period "." is included between the filename and the extension. The pathname of a file can have up to 32767 characters. For example:
>
> /usr/inventory/june1982/general.stock

The filename in this example is general.stock.

Example

```
SAVE "AUTHOR.WIL"
```

Loading the Program

If, after typing or running other programs, you wanted to go back and use this program again, you would have to "LOAD" it back into memory. Simply type: LOAD "filename", R

Example

```
LOAD "AUTHOR", R <ENTER>
```

tells MBASIC to load the program "AUTHOR" from disk into memory; optional R tells MBASIC to run it.

The SAVE and LOAD commands are discussed in more detail in Chapter 7.

Chapter 2/ Command And Execution Modes

This chapter describes MBASIC's command and execution modes. The command mode is for typing in program lines and immediate commands. The execution mode is for executing programs and immediate lines.

Command Mode

Whenever you enter the command mode, MBASIC displays a prompt:

```
Ok (prompt)
```

While you are in the command mode, MBASIC displays the prompt at the beginning of the first logical line (the line you are typing in).

A logical line is a string of up to 255 characters and is always terminated by pressing <ENTER>. A physical line, on the other hand, is one line on the display. A physical line contains a maximum of 80 characters.

For example, if you type 100 R's and then press <ENTER>, you have two physical lines, but only one logical line.

In the command mode, MBASIC does not "read" your input until you complete the logical line by pressing <ENTER>. This is called "line input", as opposed to "character input".

Interpretation of a Line

MBASIC always ignores leading spaces or tabs in the line--it jumps ahead to the first non-white character. If this character is not a digit, MBASIC treats the line as an immediate line. If it is a digit, MBASIC treats the line as a program line.

For example:

```
Ok
PRINT "THE TIME IS " TIME$ <ENTER>
```

MBASIC takes this as an immediate line.

If you type:

```
Ok
1Ø PRINT "THE TIME IS" TIME$ <ENTER>
```

MBASIC takes this as a program line.

Immediate Lines

An immediate line consists of one or more statements separated by colons. The line is executed as soon as you press <ENTER>. For example:

```
Ok
CLS: PRINT "THE SQUARE ROOT OF 2 IS" SQR(2)
```

is an immediate line. When you press <ENTER>, MBASIC executes it.

Program Lines

A program line consists of a line number in the range Ø to 65529, followed by one or more statements separated by colons. A program line may contain a maximum of 251 characters. When you press <ENTER>, the line is stored in the program text area of the memory, along with any other lines you have entered this way. The program is not executed until you type RUN or another execute command. For example:

```
1ØØ CLS: PRINT "THE SQUARE ROOT OF 2 IS" SQR(2)
```

is a program line. When you press <ENTER>, MBASIC stores it in the program text area. To execute it, type:

```
RUN  <ENTER>
```

Special Keys in the Command Mode

<BACKSPACE> or <CTRL><H>	Backspaces the cursor, erasing the preceding character in the line. This allows you typing errors.
<SPACE BAR>	Enters a blank space character and advances the cursor.

<CTRL><C> or <BREAK>	Interrupts line entry and starts over with a new line.
\ (backslash)	When followed by <ENTER> starts a new physical line without ending the current logical line.
<CAPS>	Switches the display to either all uppercase or upper/lowercase mode.
<ENTER> or <CTRL><J>	Ends the current logical line. MBASIC "takes" the line.

Execution Mode

When MBASIC is executing statements (immediate lines or programs), it is in the execution mode. In this mode, the contents of the video display are under program control.

Special Keys in the Execution Mode

<CTRL><S>	Pauses execution.
<CTRL><Q>	Continues execution.
<CTRL><C> or <BREAK>	Terminates execution and returns you to command mode.
<ENTER>	Interprets data entered from the keyboard with the INPUT statement.
<ESC>	Allows you to exit the insert command while in the MBASIC editor.

Note: The <BREAK> key has a different meaning on terminals than it does on the console. We recommend that you use <CTRL><C> instead of <BREAK> on any terminal not mentioned above.

Chapter 3/ Line Edit Mode

This mode enables you to "debug" (correct) programs quickly and efficiently. It allows you to correct a program line without having to re-type the entire line.

If MBASIC encounters a syntax error while executing a program, it automatically puts you in the "line edit mode". The display shows:

```
Syntax Error in line number
OK
line number
```

(line number represents the program line in which the error occurred.) In this case, you can use the edit mode commands and subcommands described later in this Chapter.

However, if you wish to activate the line editor yourself (because you have noticed a mistake or wish to make a change in a long program line), type:

```
EDIT line number <ENTER>
```

This lets you edit the specified line number. (If the line number you specify does not exist, an "Undefinedline number" error occurs).

You may also type:

```
EDIT .
```

The period after EDIT means that you want to edit the current program line, the last line entered, the last line altered or a line in which an error has occurred.

For example, type in and <ENTER> the following line:

```
10Ø FOR I = 1 TO 1Ø STEP .5: PRINT I, I*2, I*3: NEXT
```

(This line will be used in exercising all the edit subcommands described below).

Now type EDIT 1ØØ and press <ENTER>. MBASIC displays:

```
1ØØ
```

This starts the editor. You may now begin editing line 1ØØ.

Special Keys in the Edit Mode

<ENTER>

Pressing <ENTER> in the edit mode records all the changes you made in the current line and returns you to the command mode.

Space bar

Pressing the space bar moves the cursor over one space to the right and displays any character stored in the preceding position. For example, using line 1ØØ entered above, put MBASIC in the edit mode so the display shows:

```
1ØØ
```

Now press the space bar. The cursor moves over one space and the first character of the program line is displayed. If this character was a blank, then a blank is displayed. Press the space bar again until you reach the first non-blank character:

```
1ØØ F
```

is displayed. To move over more than one space at a time, type the desired number of spaces first, then press the space bar. For example, type 6 and press the space bar. The display should show something like this (depending on how many blanks you inserted in the line):

```
1ØØ FOR I =
```

Now type 8 and press the space bar. The cursor moves over eight spaces to the right, and eight more characters are displayed.

```
1ØØ FOR I = 1 TO 1Ø
```

L (List Line)

displays the remainder of the program line (unless MBASIC is under one of the insert subcommands listed below). The cursor drops down to the next line of the display, reprints the current line number, and moves to the first position of the line.

For example, when the display shows

```
10Ø
```

press L (without pressing <ENTER> key). Line 10Ø is displayed:

```
10Ø FOR I = 1 TO 1Ø STEP .5: PRINT I, 1*2, I*3: NEXT
10Ø
```

This lets you look at the line in its current form while you're doing the editing.

Insert Subcommand Mode

The insert subcommand mode allows you to add material to a line while editing it. The three keys you can use to enter this subcommand mode are X, I and H.

X (Extend Line)

Displays the rest of the current line. Typing X also moves the cursor to the end of the line and puts MBASIC in the insert subcommand mode. This enables you to add material to the end of the line.

For example, using line 10Ø, when the display shows

```
10Ø
```

press X (without pressing <ENTER>) and the entire line is displayed; notice that the cursor now follows the last character on the line:

```
10Ø FOR I = 1 TO 1Ø STEP .5: PRINT I, I*2, I*3: NEXT
```

We can now add another statement to the line, or delete material from the line by using the <BACKSPACE> key. For example, type

```
: PRINT "DONE"
```

at the end of the line. Now press <ENTER>. If you type LIST 1ØØ, the display should show something like this:

```
1ØØ FOR I = 1 TO 1Ø STEP .5: PRINT I, I*2, I*3: NEXT: PRINT
    "DONE"
```

Note: If you want to continue editing the line, press <ESC> to get out of the insert subcommand mode.

I (Insert)

Inserts material beginning at the current cursor position on the line.

For example, type and <ENTER> the EDIT 1ØØ command, then use the space bar to move over to the decimal point in line 1ØØ. The display shows:

```
1ØØ FOR I = 1 TO 1Ø STEP .
```

Suppose you want to change the increment from .5 to .25. Press the I key (don't press <ENTER>) and MBASIC now lets you insert material at the current position. Type 2 now, and the display shows:

```
1ØØ FOR I = 1 TO 1Ø STEP .2
```

You have made the necessary change, so press <ESC> to escape from the insert subcommand. Now press the L key to display the remainder of the line and move the cursor back to the beginning of the line:

```
1ØØ FOR I = 1 TO 1Ø STEP .25: PRINT I, I*2, I*3: NEXT: PRINT
    "DONE"
1ØØ
```

Note: You can also exit the insert subcommand and save all changes by pressing <ENTER>. This returns you to command mode.

H (Hack and Insert)

Deletes the remainder of a line and lets you insert material at the current cursor position.

For example, using line 1ØØ, enter the edit mode and space over until just before the PRINT "DONE" statement. Suppose you wish to delete this statement and insert an END statement. The display shows:

```
100 FOR I = 1 TO 10 STEP .25: PRINT I, I*2, I*3: NEXT:
```

Type H, then type END. Press <ENTER>. List the line:

```
100 FOR I = 1 TO 10 STEP .25: PRINT I, I*2, I*3: NEXT: END
```

should be displayed.

Note: To continue editing the line, press <ESC> this takes you out of the insert subcommand mode.

A (Cancel and Restart)

Moves the cursor back to the beginning of the program line and cancels editing changes already made.

For example, if you have added, deleted, or changed something in a line, and you wish to go back to the beginning of the line and cancel the changes already made: first press <ESC> (to escape from any subcommand you may be executing); then press A. The cursor drops down to the next line, displays the line number and moves to the first character position.

E (Save Changes and Exit)

Ends editing and saves all changes made. You must be in edit mode, not executing any subcommand, when you press E to end editing.

Q (Cancel and Exit)

Ends editing and cancels all changes made in the current editing session. If you've decided not to change the line, type Q to cancel changes and leave the edit mode.

If a syntax error is detected during program execution, MBASIC starts the editor. To examine variable values, you must press Q or <ENTER> before typing any other command.

nD (Delete)

Deletes the specified number n of characters to the right of the cursor. The deleted characters appear enclosed in backslashes.

For example, using line 10Ø, space over to just before the PRINT statement:

```
10Ø FOR I = 1 TO 1Ø STEP .25:
```

Now type 19D. This tells MBASIC to delete 19 characters to the right of the cursor. The display should show something like this:

```
10Ø FOR I = 1 TO 1Ø STEP .25: \PRINT I, I*2, I*3:\NEXT: END
```

When you list the complete line, you will see that everything from the PRINT to the next statement has been deleted.

nC (Change)

Lets you change the specified number of characters beginning at the current cursor position. If you type C without a preceding number, MBASIC assumes you want to change one character. When you have entered n number of characters, MBASIC returns you to the edit mode (so you're not in the nC subcommand).

For example, using line 10Ø, suppose you want to change the final value of the FOR-NEXT loop, from "1Ø" to "15". In the edit mode, space over to just before the "Ø" in "1Ø".

```
10Ø FOR I = 1 TO 1
```

Now type C. MBASIC assumes you want to change just one character. Press 5, then press L. When you list the line, you'll see that the change has been made.

```
10Ø FOR 1 = 1 TO 15 STEP .25: NEXT: END
```

would be the current line if you've followed the editing sequence in this chapter.

nSc (Search)

Searches for the nth occurrence of the character c, and moves the cursor to that position. If you don't specify a value for n, MBASIC searches for the first occurrence of the specified character. If character c is not found, cursor goes to the end of the line.

Note: MBASIC only searches through characters to the right of the cursor.

For example, using the current form of line 100, type EDIT 100 <ENTER>. then press 2S:. This tells MBASIC to search for the second occurrence of the colon character. The display should show:

```
100 FOR I = 1 TO 15 STEP .25: NEXT
```

You may now execute one of the subcommands beginning at the current cursor position. For example, suppose you want to add the counter variable after the NEXT statement. Type I to enter the insert subcommand, then press the spacebar to insert a space and type the variable name, I. That's all you want to insert, so press <ESC> to escape from the insert subcommand mode. The next time you list the line, it should appear as:

```
100 FOR I = 1 TO 15 STEP .25: NEXT I: END
```

nKc (Search and Kill)

Deletes all characters up to the nth occurrence of character c, and moves the cursor to that position.

For example, using the current version of line 100, suppose we wanted to delete the entire line up to the END statement. Type EDIT 100 <ENTER>, then type 2K:. This tells MBASIC to delete all characters up to the 2nd occurrence of the colon.

```
100 \FOR I = 1 TO 15 STEP .25: NEXT I\
```

should be displayed. The second colon still needs to be deleted, so type D. The display now shows:

```
100 \FOR I = 1 TO 15 STEP .25: NEXT I\\:\
```

Press <ENTER> and type LIST 100 <ENTER>

Line 100 should look something like this:

```
100 END
```

n <BACKSPACE>

Moves the cursor to the left by n spaces. If no number n is given, the cursor moves back one space. When the cursor backspaces, all characters in its path are displayed in a backwards format, but they are not deleted from the program. Use the space bar to advance the cursor forward and re-display the characters.

BASIC Language

MBASIC Reference Manual

SECTION II/ The MBASIC Language

Chapter 4/ MBASIC Concepts

This chapter describes how to use the full power of TRS-XENIX MBASIC. This information can help programmers build powerful and efficient programs. If you are still something of a novice, you might want to skip this chapter for now, keeping in mind that the information is here when you need it.

The chapter is divided into four sections:

A. Overview - Elements of a Program. This section defines many of the terms we will be using in the chapter.

B. How MBASIC Handles Data. Here we discuss how MBASIC classifies and stores data. This shows you how to get MBASIC to store your data in its most efficient format.

C. How MBASIC Manipulates Data. This gives you an overview of all the different operators and functions you can use to manipulate and test your data.

D. How to Construct an Expression. Understanding this topic will help you in constructing powerful statements instead of using many short ones.

Overview - Elements of a Program

This overview defines the elements of a program. A program is made up of "statements"; statements may have several "expressions."

We will refer to these terms during the rest of this chapter.

Program

A program is made up of one or more numbered lines. Each line contains one or more MBASIC statements. MBASIC allows line numbers from Ø to 65529 inclusive. You may include up to 251 characters per line, not including the line number (see note below). You may also have two or more statements to a line, separated by colons.

Note: MBASIC allows you to input more than 251 characters per line; however, unpredictable results can occur if the number of characters exceeds 251.

Each program can hold 16,777,215 bytes of data. This includes the user program, variables and strings. However, you are limited by the actual amount of memory in MBASIC. Refer to your current operators manual for this information.

Here is a sample program:

Line number	MBASIC statement	Colon between statements	MBASIC statement

```
1ØØ CLS: FOR I = 1 TO 15
11Ø PRINT "NUMBER ";I
12Ø FOR J = 1 TO 5ØØ: Next J
13Ø NEXT I
14Ø END
```

When MBASIC executes a program, it handles the statements one at a time, starting with the first and proceeding to the last. Some statements, such as GOTO, ON...GOTO, GOSUB, change this sequence.

Statements

A statement is a complex instruction to MBASIC, telling MBASIC to perform specific operations. For example:

```
GOTO 1ØØ
```

Tells MBASIC to perform the operations of (1) locating line 100 and (2) executing the statement on that line.

END

Tells MBASIC to perform the operation of ending execution of the program.

Many statements instruct MBASIC to perform operations with data. For example, in the statement:

PRINT"SEPTEMBER REPORT"

the data is SEPTEMBER REPORT. The statement instructs MBASIC to print the data inside quotes.

Line Termination

The line terminator for ASCII files under the TRS-XENIX operating system is the new line character. Under TRS-XENIX the new line character is represented by a carriage return or a line feed. Either a carriage return or a line feed will terminate a BASIC line.

Line Continuation

Continuing a single logical line across several physical lines is done by preceding the line terminator with the backslash character "\". For example:

```
IF NAME = "SARA"  \
   THEN PRINT "TRUE" \
   ELSE PRINT "FALSE" \
```

The backslashes were typed before the carriage returns were typed, thus causing the 3 physical lines to appear to BASIC as a single logical line.

A backslash placed in any other position will have its normal meaning as an integer divide. The internal representation for line continuation used in binary files remains the same for all versions of TRS-XENIX BASIC.

Expressions

An expression is actually a general term for data. There are four types of expressions:

1. Numeric expressions, which are composed of numeric data.

Examples:

```
(1 + 5.2)/3
D
5*B
3.7682
ABS(X) + RND(Ø)
SIN(3 + E)
```

2. String expressions, which are composed of character data.

Examples:

```
A$
"STRING"
"STRING" + "DATA"
MO$ + "DATA"
MID$(A$,2,5) + MID$("MAN",1,2)
M$ + A$ + B$
```

3. Relational expressions, which test the relationship between two expressions.

Examples:

```
A = 1
A$>B$
```

4. Logical expressions, which test the logical relationship between two expressions.

Examples:

```
A$ = "YES" AND B$ = "NO"
c>5 OR M<B OR Ø>2
578 AND 452
```

Functions

Functions are automatically subroutines. Most MBASIC functions perform computations on data. Some serve a special purpose, such as controlling the video display or providing data on the status of MBASIC. You may use functions in the same manner that you use any data: as part of a statement.

These are some of MBASIC's functions:

```
INT
ABS
STRING$
```

For example, ABS returns the absolute value of a numeric expression. The following example shows how this function works:

```
PRINT ABS(7*(-5)) <ENTER>
 35
Ok
```

How MBASIC Handles Data

MBASIC offers several different methods of handling your data. Using these methods properly can greatly improve the efficiency of your program. In this section we discuss:

1. Ways of Representing Data
   a. Constants
   b. Variables
2. How MBASIC Stores Data
   a. Numeric (integer, single precision, double precision)
   b. String
3. How MBASIC Classifies Constants
4. How MBASIC Classifies Variables
5. How MBASIC Converts Data

Ways of Representing Data

MBASIC recognizes data in two forms: directly (as constants), or by reference to a memory location (as variables).

Constants

All data is input into a program as "constants" - values which are not subject to change. For example, the statement:

```
PRINT "1 PLUS 1 EQUALS"; 2
```

contains one string constant,

```
1 PLUS 1 EQUALS
```

and one numeric constant

```
2
```

In these examples, the constants "input" to the PRINT statement. They tell PRINT what data to print on the display.

These are more examples of constants:

```
3.14159              "L.O.SMITH"
1.775E+3             "Ø123456789ABCDEF"
"NAME TITLE"         -123.45E-8
57                   "AGE"
```

Variables

A variable is a place in memory where data is stored. Unlike a constant, a variable's value can change. This allows you to write programs dealing with changing quantities. For example, in the statement:

```
A$ = "OCCUPATION"
```

The variable A$ now contains the data OCCUPATION. However, if this statement appeared later in the program:

```
A$ = "FINANCE"
```

The variable A$ would no longer contain OCCUPATION. It would now contain the data FINANCE.

Variable Names

In MBASIC, variables are represented by names. Variable names must begin with a letter, A through Z. This letter may be followed by one or more characters (digits or letters).

For example:

AM A A1 BALANCE EMPLOYEE2

are all valid and distinct variable names.

Variable names may be of any length up to 40 characters.

Reserved Words

Certain combinations of letters are reserved as MBASIC keywords and operator names. These combinations cannot be used in variable names.

For example:

OR LEN OPTION

cannot be used as variable names.

But keywords may appear inside variable names. For example:

BORE STRLEN OPTIONC

See the Appendix B on MBASIC's reserved words.

Simple and Subscripted Variables

Variables may also be "subscripted" so that an entire list of data can be stored under one variable name. This method of data storage is called an array. For example, an array named A may contain these elements (subscripted variables):

A(0) A(1) A(2) A(3) A(4)

You may use each of these elements to store a seperate data item, such as:

```
A(0) = 5.3
A(1) = 7.2
```

```
A(2) = 8.3
A(3) = 6.8
A(4) = 3.7
```

In this example, array A is a one-dimensional array, since each element contains only one subscript. An array may also be two-dimensional, with each element containing two subscripts. For example, a two-dimensional array named X could contain these elements:

```
X(Ø,Ø) = 8.6          X(Ø,1) = 3.5
X(1,Ø) = 7.3          X(1,1) = 32.6
```

With MBASIC, you may have as many dimensions in your array as you would like. Here is an example of a three-dimensional array named L which contains these eight elements:

```
L(Ø,Ø,Ø) = 35233       L(Ø,1,Ø) = 96522
L(Ø,Ø,1) = 52ØØØ       L(Ø,1,1) = 1Ø255

L(1,Ø,Ø) = 33333       L(1,1,Ø) = 96253
L(1,Ø,1) = 53853       L(1,1,1) = 79654
```

Each index of the array is limited to the range Ø to 32767. MBASIC assumes that all arrays contain 11 elements in each dimension. If you want more elements you must use the DIM statement at the beginning of your program to dimension the array.

For example, to dimension array L, put this line at the beginning of the program:

```
DIM(1,1,1)
```

to allow room for two elements in the first dimension; two in the second, and two in the third for a total of 2*2*2 = 8 elements.

Arrays are discussed later in this chapter.

How MBASIC Stores Data

The way that MBASIC stores data determines the amount of memory it consumes and the speed in which MBASIC can process it.

Numeric Data

You may get MBASIC to store all numbers in your program as either integer, single precision, or double precision. In deciding how to get MBASIC to store your numeric data, remember the tradeoffs. Integers are the most efficient and the least precise. Double precision is the most precise and least efficient. For information on the internal representation of numbers refer to Appendix H.

Integers
(Speed and Efficiency, Limited Range)

To be stored as an integer, a number must be whole and in the range of -32768 to 32767. An integer value requires only two bytes of memory for storage. Arithmetic operations are faster when both operands are integers.

For example:

1 3200 -2 500 -12345

can all be stored as integers.

Single Precision
(General Purpose, Full Numeric Range)

Single precision numbers can include up to six significant digits, and can represent normalized values* with exponents up to 64. Refer to Appendix H for more information on the range of single precision numbers.

A single precision value requires four bytes of memory for storage. MBASIC assumes a number is double precision if you do not specify the level of precision.

* In this manual, normalized value is one in which exactly one digit appears to the left of the decimal point. For example, 12.3 expressed in normalized form is 1.23 X 10.

For example:

10.001 -200034 1.774E6 6.024E-23 123.456

can all be stored as single precision values.

Note: When used in a decimal number, the symbol E stands for "single precision times 1Ø to the power of..." Therefore 6.Ø24E-23 represents the single precision value:

$$6.Ø24 \times 1Ø^{-23}$$

Double Precision
(Maximum Precision, Slowest in Computations)

Double precision numbers can include up to 14 significant digits, and can represent values in the same range as that for single precision numbers. A double precision value requires eight bytes of memory for storage.

For example:

1Ø1Ø234578
-8.7777651Ø1Ø
3.141592653589
8.ØØ1ØØ7Ø8D12

can all be stored as double precision values.

Note: When used in a decimal number, the symbol D stands for "double precision times 1Ø to the power of..." Therefore 8.ØØ1ØØ7Ø8D12 represents the value

$$8.ØØ1ØØ7Ø8 \times 1Ø^{12}$$

The default type for a variable is double precision.

For more information on Numeric data refer to Appendix H.

Strings

Strings (sequences of characters) are useful for storing non-numeric information such as names, addresses, text, etc. You may store any ASCII characters as a string.

For example, the data constant:

Jack Brown, Age 38

can be stored as a string of 18 characters. Each character (and blank) in the string is stored as an ASCII code, requiring one byte

of storage. Five bytes of overhead are added to the present contents of the string.

MBASIC would store the above string constant internally as:

Hex Code	4A	61	63	6B	20	42	72	6F	77	6E	2C	20	41	67	65	20	33	38
ASCII Character	J	a	c	k		B	r	o	w	n	,		A	g	e		3	8

A string can be up to 32767 characters long. Strings with length zero are called "null" or "empty".

How MBASIC Classifies Constants

When MBASIC encounters a data constant in a statement, it must determine the type of the constant: string, integer, single precision, or double precision. First, we will list the rules MBASIC uses to classify the constant. Then we will show you how you can override these rules, if you want a constant stored differently:

Rule 1

If the value is enclosed in double-quotes, it is a string. For example:

```
"YES"
"3331 Waverly Way"
"123456789Ø"
```

are all classified as strings.

Rule 2

If the value is not in quotes, it is a number. (An exception to this rule is during data input by an operator, and in DATA lists. See INPUT, INKEY$, and DATA)

For example:

```
123ØØ1
1
-7.3214E + 6
```

are all numeric data.

Rule 3

Whole numbers in the range of -32768 to 32767 are integers. For example:

```
12350
-12
1ØØ12
```

are integer constants.

Rule 4

If the number is not an integer and contains six or fewer digits, it is single precision. For example:

```
123456
-1.23
1.3321
```

are all classifed as single precision.

Rule 5

If the number contains more than six digits, it is double precision. For example, these numbers:

```
12345678Ø1234
-1ØØØØØØØØØØØØ.1
2.777ØØØ321
```

are all classified as double precision.

Type Declaration Tags

You can override MBASIC's normal typing criteria by adding the following "tags" to the end of the numeric constant:

! Makes the number single precision. For example, in the statement:

```
A = 12.345678Ø1234!
```

the constant is classified as single precision, and shortened to six digits: 12.3457.

E Single precision exponential format. The E indicates the constant is to be multiplied by a specific power of 1Ø. For example:

```
A = 1.2E5
```

stores the single precision number 12ØØØØ in A.

Makes the number double precision. For example, in statement:

```
PRINT 3#/7
```

the first constant is classified as double precision before the division takes place.

D Double precision exponential format. The D indicates the constant is to be multiplied by a specified power of 1Ø. For example:

```
A = 1.23456789D-1
```

The double precision constant has the value Ø.123456789.

How MBASIC Classifies Variables

When MBASIC encounters a variable name in the program, it classifies it as either a string, an integer, a single precision number, or a double precision number.

MBASIC classifies all variable names as double precision initially. For example:

```
AB        AMOUNT        XY        L
```

are all double precision initially. If this is the first line of your program:

```
LP = 1.2
```

MBASIC classifies LP as a double precision variable.

However, you may assign different attributes to variables by using definition statements at the beginning of your program:

DEFSTR - Defines variables as string.
DEFINT - Defines variables as integer.
DEFSNG - Defines variables as single precision.
DEFDBL - Defines variables as double precision. (Since MBASIC classifies all variables as double precision initially, you would only need to use DEFDBL if one of the other DEF statements were used.

For example:

```
DEFSTR L
```

makes MBASIC classify all variables which start with L as string variables. After this statement, the variables:

L LP LAST

can all hold string values only.

Type Declaration Tags

As with constants, you can always override the type of a variable name by adding a type declaration tag at the end. There are four type declaration tags for variables:

% Integer
! Single precision
Double precision
$ String

For example:

I% FT% NUM% COUNTER%

are all integer variables, **regardless** of what attributes have been assigned to the letters I, F, N, and C.

T! RY! QUAN! PERCENT!

are all single precision variables, **regardless** of what attributes have been assigned to the letters T, R, Q, and P.

```
X#        RR#       PREV#     LSTNUM#
```

are all double precision variables, **regardless** of what attributes have been assigned to the letters X, R, P, and L.

```
Q$        CA$       WRD$      ENTRY$
```

are all string variables, **regardless** of what attributes have been assigned to the letters Q, C, W, and E.

Note that any given variable name can represent four different variables. For example:

```
A5#       A5!       A5%       A5$
```

are all valid and **distinct** variable names.

One further implication of type declaration: Once you define a type declaration tag it has no effect on that particular variable name. For example, after the statement:

```
DEFSTR C
```

the variable referenced by the name C1 is identical to the variable referenced by the name C1$.

How MBASIC Converts Numeric Data

Often your program might ask MBASIC to assign one type of constant to a different type of variable. For example:

```
A% = 2.34
```

In this example, MBASIC must first convert the single precision constant 2.34 to an integer in order to assign it to the integer variable A%.

You might also want to convert one type of variable to a different type, such as:

```
A# = A%
A! = A#
A! = A%
```

The conversion procedures are listed on the following pages.

Single or double precision to integer type

MBASIC rounds the fractional portion of the number.

Note: The original value must be greater than or equal to -32768, and less than 32768.

Examples

```
A% = 32766.9
```

assigns A% the value 32767.

```
A% = 2.5D3
```

assigns A% the value 2500.

```
A% = -123.456789Ø1234578
```

assigns A% the value -123.

```
A% = -32767.5
```

assigns A% the value -32768.

Integer to single or double precision

The converted value looks like the original value with zeros to the right of the decimal place.

Examples

```
A# = 32767
```

Stores 32767.0000000000 in A# and would print 32767.

```
A! = -1234
```

Stores -1234.00 in A! and would print -1234.

Double to single precision

This involves converting a number with up to 14 significant digits into a number with no more than six. MBASIC rounds off the least siginificant digits to produce a six-digit number.

Examples

```
A! = 1.23456789Ø124567
```

stores 1.23457 in A!. The statement:

```
PRINT A
```

displays the value 1.23457.

```
A! = 1.3333333333333
```

stores 1.33333 in A!.

Single to double precision

To make this conversion, MBASIC simply adds trailing zeros to the single precision number. For example:

```
A# = 1.5
```

stores 1.5ØØØØØØØØØØØØØ in A# and would print 1.5.

Illegal Conversions

MBASIC cannot automatically convert numeric values to string, or vice versa. For example, the statements:

```
A$ = 1234
A% = "1234"
```

are illegal. They would return a "Type mismatch" error. (Use STR$ and VAL to accomplish such conversions.)

How MBASIC Manipulates Data

You have many fast methods you may use to get MBASIC to count, sort, test, and rearrange your data. These methods fall into two categories:

1. Operators
   a. numeric
   b. string
   c. relational
   d. logical

2. Functions

Operators

An operator is the single symbol or word which signifies some action to be taken on either one or two specified values referred to as operands.

In general, an operator is used like this:

```
operand-1  operator  operand-2

    6         +          2
```

The addition operator + connects or relates its two operands, 6 and 2, to produce the result 8.

Operand-1 and -2 can be expressions.

A few operations take only one operand, and are used like this:

```
operator   operand

   -          5
```

The negation operator - acts on a single operand 5 to produce the result negative 5.

Neither 6 + 2 or -5 can stand alone; they must be used in statements to be meaningful to MBASIC. For example:

```
A = 6 + 2
PRINT -5
```

Operators fall into four categories:

- Numeric
- String
- Relational
- Logical

based on the kinds of operands they require and the results they produce.

Numeric Operators

Numeric Operators are used in numeric expressions. Their operands must always be numeric, and the results they produce are one numeric data item.

In the description below, we use the terms integer, single precision, and double precision operations. Integer operations involve two-byte operands, single precision operations involve four-byte operands, and double precision operations involve eight-byte operands. The more bytes involved, the slower the operation.

There are five different numeric operators. Two of them, sign + and sign -, are unary, that is, they have only one operand. A sign operator has no effect on the precision of its operand.

For example, in the statement:

```
PRINT -77, +77
```

the sign operators - and + produce the values negative 77 and positive 77, respectively.

Note: When no sign operator appears in front of a numeric term, + is assumed.

The other numeric operators are all binary, that is, they all take two operands.

These operators are, in order of precedence:

^	Exponentiation
*, /	Multiplication, Division
+,-	Addition, Subtraction

Exponentiation

The symbol ^ denotes exponentiation. It converts both its operands to double precision, and returns a double precision result.

For example:

```
PRINT 6^.3
```

prints 6 to the .3 power.

Multiplication

The * operator is the symbol for multiplication. Once again, the operation is done with the precision of the more precise operand (the less precise operand is converted).

Examples:

```
PRINT 33 * 11
```

integer multiplication is performed.

```
PRINT 33 * 11.1
```

double precision multiplication is performed.

```
PRINT 12.345678Ø1234 * 11
```

double precision multiplication is performed.

Division

The / symbol is used to indicate ordinary division. Both operands are converted to single precision or double precision, depending on their original precision:

. If either operand is double precision, then both are converted to double precision and eight-byte division is performed.

Examples:

```
PRINT 3/4
```

double precision division is performed.

```
PRINT 3.8/4
```

double precision division is performed.

```
PRINT 3/1.2345678Ø1234
```

double precision division is performed.

Addition

The + operator is the symbol for addition. The addition is done with the precision of the most precise operand (the less precise operand is converted).

For example, when one operand is integer type and the other is single precision, the integer is converted to single precision and four-byte addition is performed. When one operand is single precision and the other is double precision, the single precision number is converted to double precision and eight-byte addition is performed.

Examples:

```
PRINT 2 + 3
```

integer addition is performed.

```
PRINT 3.1 + 3
```

double precision addition is performed.

```
PRINT 1.2345678Ø1234 + 1
```

double precision addition is performed.

Subtraction

The - operator is the symbol for subtraction. As with addition, the operation is done with the precision of the most precise operand (the less precise operand is converted).

Examples:

```
PRINT 33 - 11
```

integer subtraction is performed.

```
PRINT 33 - 11.1
```

double precision subtraction is performed.

```
PRINT 12.345678Ø1234 - 11
```

double precision subtraction is performed.

String Operator

MBASIC has a string operator (+) which allows you to concatenate (link) two strings into one. This operator should be used as part of a string expression. The operands are both strings and the resulting value is one piece of string data.

The + operator links the string on the right of the sign to the string on the left. For example:

```
PRINT "CATS" + "LOVE" + "MICE"
```

prints:

```
CATSLOVEMICE
```

Since MBASIC does not allow one string to be longer than 32767 characters, you will get an error if your resulting string is too long.

Relational Operators

Relational operators compare two numeric or two string expressions to form a relational expression. This expression reports whether the comparison you set up in your program is true or false. It returns a -1 if the relation is true; a Ø if it is false.

Numeric Relations

This is the meaning of the operators when you use them to compare numeric expressions:

<	Less than
>	Greater than
=	Equal to

<> or ><	Not equal to
=< or <=	Less than or equal to
=> or >=	Greater than or equal to

Examples of true relational expressions:

```
1 < 2
2 <> 5
2 <= 5
2 <= 2
5 > 2
7 = 7
```

String Relations

The relational operators for string expressions are the same as above, although their meanings are slightly different. Instead of comparing numerical magnitudes, the operators compare their ASCII sequence. This allows you to sort string data:

<	Precedes
>	Follows
>< or <>	Does not have the same precedence
<=	Precedes or has the same precedence
>=	Follows or has the same precedence

MBASIC compares the string expressions on a character-by-character basis from left to right. When it finds a non-matching character, it checks to see which character has the lowest ASCII code. The character with the lowest ASCII code is the smallest (precedent) of the two strings.

Examples of true relational expressions:

```
"A" < "B"
```

The ASCII code for A is decimal 65; for B it's 66.

```
"CODE" < "COOL"
```

The ASCII code for O is 79; for D it's 68.

If while making the comparision, MBASIC reaches the end of one string before finding non-matching characters, the shorter string is the precedent. For example:

```
"TRAIL" < "TRAILER"
```

Leading and trailing blanks are significant. For example:

```
" A" < "A"
```

ASCII for the space character is 32; for A, it's 65.

```
"Z-8Ø" < "Z-8ØA"
```

The string on the left is four characters long; the string on the right is five.

How to Use Relational Expressions

Normally, relational expressions are used as the test in an IF/THEN statement. For example:

```
IF A = 1 THEN PRINT "CORRECT"
```

MBASIC tests to see if A is equal to 1. If it is, MBASIC prints the message "CORRECT".

```
IF A$ < B$ THEN 5Ø
```

if string A$ alphabetically precedes string B$, then the program branches to line 5Ø.

```
IF R$ = "YES" THEN PRINT A$
```

if R$ equals YES then the message stored as A$ is printed.

However, you may also use relational expressions simply to return the true or false results of a test. For example:

```
PRINT 7 = 7
```

prints - 1 since the relation tested is true.

```
PRINT "A" > "B"
```

prints Ø because the relation tested is false.

Logical Operators

Logical operators make logical comparisions. Normally, they are used in IF/THEN statements to make a logical test between two or more relations. For example:

```
IF A = 1       OR C = 2       THEN PRINT X
```

The logical operator, OR, compares the two relations A = 1 and C = 2.

Logical operators may also be used to make bit comparisions of two numeric expressions.

For this application, MBASIC does a bit-by-bit comparision of the two operands, according to predefined rules for the specific operator.

Note: The operands are converted to integer type, stored internally as 16-bit, two's complement numbers. To understand the results of bit-by-bit comparisions, you need to keep this in mind.

The following table summarizes the action of Boolean operators in bit manipulation.

Operator	Meaning of Operation	First Operand	Second Operand	Result
AND	When both bits are 1, the results will be 1. Otherwise, the result will be Ø.	1	1	1
		1	Ø	Ø
		Ø	1	Ø
		Ø	Ø	Ø
OR	Result will be 1 unless both bits are Ø.	1	1	1
		1	Ø	1
		Ø	1	1
		Ø	Ø	Ø
NOT	Result is opposite of bit.	1		Ø
		Ø		1
XOR	Result will be 1 when both bits are opposite.	1	1	Ø
		1	Ø	1
		Ø	1	1
		Ø	Ø	Ø

EQV	Result will be 1 when both bits are the same.	1 1 Ø Ø	1 Ø 1 Ø	1 Ø Ø 1
IMP	Result will be 1 unless the first bit is 1 and the second bit is zero.	1 1 Ø Ø	1 Ø 1 Ø	1 Ø 1 1

Hierarchy of Operators

When your expressions have multiple operators, MBASIC performs the operations according to a well-defined hierarchy, so that results are always predictable.

Parentheses

When a complex expression includes parentheses, MBASIC always evaluates the expressions inside the parentheses before evaluating the rest of the expression. For example, the expression:

```
8 - (3 - 2)
```

is evaluated like this:

```
3 - 2 = 1
8 - 1 = 7
```

With nested parentheses, MBASIC starts evaluating the innermost level first and works outward. For example:

```
4 * (2 - (3 - 4))
```

is evaluated like this:

```
3 - 4 = -1
2 - (-1) = 3
4 * 3 = 12
```

Order of Operations

When evaluating a sequence of operations on the same level of parenthesis, MBASIC uses a hierarchy to determine what operation to do first.

The two listings below show the hierarchy MBASIC uses. Operators are shown in decreasing order of precedence and are executed as encountered **from left to right:**

For Numeric Operations:

```
          ^ or (Exponentiation)
+,- (Unary sign operands [not addition and
     subtraction])

     *,/ (Multiplication and division)

     +,- (Addition and subtraction)
          <,>,=,<=,>=,<>

               NOT
               AND
               OR
               XOR
               EQV
               IMP
```

For String Operations:

```
+
<,>,=,<=,>=,<>
```

For example, in the line:

```
X*X + 5^2.8
```

MBASIC finds the value of 5 to the 2.8 power. Next it multiplies X*X, and finally adds the value of 5 to the 2.8. If you want MBASIC to perform the indicated operations in a different order, you must add parentheses. For example:

```
(X*(X + 5))^2.8
```

or

```
X*(X + 5) ^2.8
```

Here's another example:

```
IF X = Ø  OR  Y > Ø  AND  Z = 1  THEN  255
```

The relational operators = and > have the highest precedence, so MBASIC performs them first, one after the next, from left to right. Then the logical operations are performed. AND has a higher precedence than OR, so MBASIC performs the AND operation before OR.

If the above line looks confusing because you can't remember which operator is precedent over which, then you can use parentheses to make the sequence obvious:

```
IF X = Ø  OR  ((Y > Ø)  AND  (Z = 1))  THEN 255
```

Functions

A function is a built-in sequence of operations which MBASIC performs on data. MBASIC functions save you from having to write a MBASIC routine, and they operate faster than a MBASIC routine would.

Examples:

```
SQR (A + 6)
```

tells MBASIC to compute the square root of (A + 6).

```
MID$ (A$,3,2)
```

tells MBASIC to return a substring of the string A$, starting with the third character, with a length of 2.

MBASIC functions are described in more detail in Chapter 7.

If the function returns numeric data, it is a numeric function and may be used in a numeric expression. If it returns string data, it is a string function and may be used in a string expression.

How to Construct an Expression

Understanding how to construct an expression will help you put together powerful statements,instead of using many short ones. In this section we will discuss the two kinds of expressions you may construct:

- Simple
- Complex

as well as how to construct a function.

As we have stated before, an expression is actually data. This is because once MBASIC performs all the operations, it returns one data item. An expression may be string or numeric. It may be composed of:

- Constants
- Variables
- Operators
- Functions

Expressions may be either simple or complex:

A **simple expression** consists of a single term: a constant, variable or function. If it is a numeric term, it may be preceded by an optional + or - sign, or by the logical operator NOT.

For example:

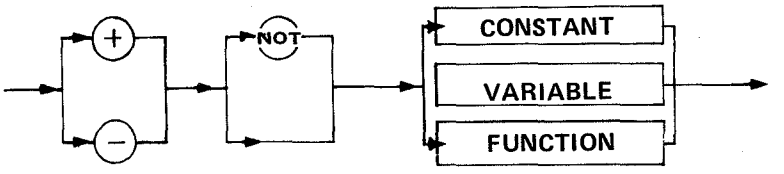
```
+A   3.3  -5   SQR(8)
```

are all simple numeric expressions, since they only consist of one numeric term.

```
A$   STRING$ (2Ø,A$)     "WORD"    "M"
```

are all simple string expressions, since they only consist of one string term.

Here's how a **simple expression** is formed:

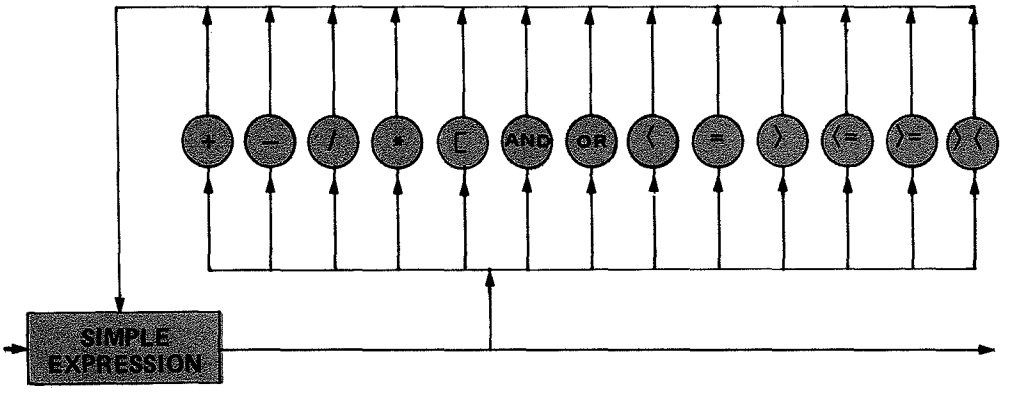


A **complex expression** consists of two or more terms (simple expressions) combined by operators. For example:

A-1 X+3.2-Y 1=1 A AND B ABS(B)+LOG(2)

are all examples of complex numeric expressions. (Notice that you can use the relational expression (1=1) and the logical expression (A AND B) as a complex numeric expression since both actually return numeric data.)

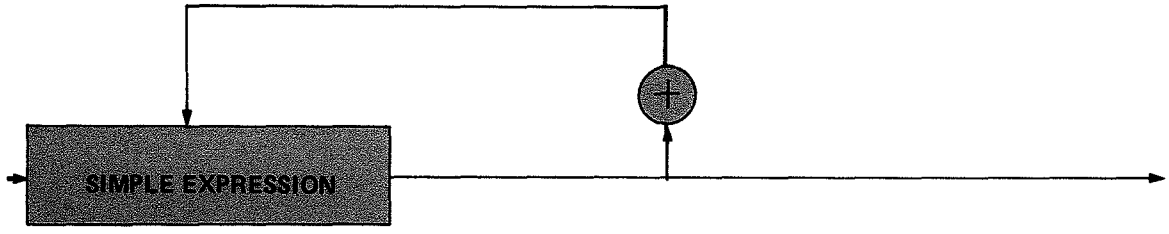
A **complex numeric expression** is formed using the following operators:



A$ + B$ "Z" + Z$ STRING$(1Ø, "A") + "M"

are all examples of complex string expressions.

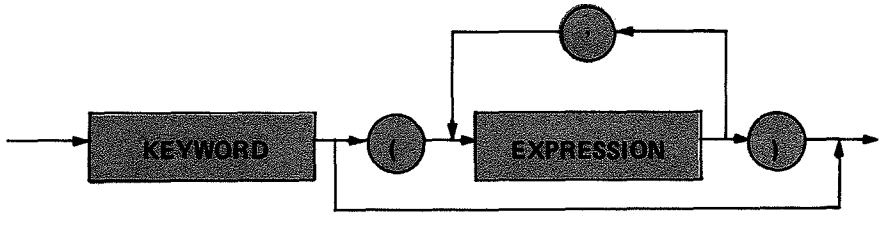
A **complex string expression** is formed using the following operator:



Most functions, except functions returning system information, require that you input either or both of the following kinds of data:

- One or more numeric expressions
- One or more string expressions

This is how a **function** is formed:



If the data returned is a number, the function may be used as a term in a numeric expression. If the data is a string, the function may be used as a term in a string expression.

SIN(A) STR$(X) VAL(A) LOG(.53)

are all examples of functions.

Chapter 5/ Disk Files

You may want to store data on your disk for future use. To do this, you need to store the data in a "disk file". A disk file is an organized collection of related data. It may contain a mailing list, a personnel record, or almost any kind of information. This is the largest block of information on disk that you can address with a single command.

To transfer data from a MBASIC program to a disk file, and vice-versa, the data must first go through a "buffer". This is an area in memory where data is accumulated for further processing.

With MBASIC, you can create and access two types of disk files. The difference between these two types is that each is created in a different "mode". The mode you choose determines what kind of access you will have to the file: sequential access or direct access.

Sequential-Access Files

With a sequential-access file, you can only access data in the same order it was stored: sequentially. To read from or write to a particular section in the file, you must first read through all the contents in the file until you get to the desired section.

Data is stored in a sequential file as ASCII characters. Therefore, it is ideal for storing free-form data without wasting space between data items. However, it is limited in flexibility and speed.

The statements and functions used with sequential files are:

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

These statements and functions are discussed in more detail in Chapters 6 and 7.

Creating a Sequential-Access File

1. To create the file, OPEN it in "O" (output) mode and assign it a buffer number (1 to 255).

   Example

   ```
   OPEN "O", 1, "LIST.EMP"
   ```

   opens a sequential output file named LIST/EMP and gives buffer 1 access to this file.

2. To write data to the file, use the WRITE# statement (you can also use PRINT#).

   Example

   ```
   WRITE# 1, N$
   ```

   writes variable N$ to the file, using buffer 1 (the buffer used to OPEN the file). Remember that data must go through a buffer before it can be written to a file.

3. To ensure that all the data was written to the file, use the CLOSE statement.

   Example

   ```
   CLOSE 1
   ```

   closes access to the file, using buffer 1 (the same buffer used to OPEN the file).

   Sample Program

   ```
   1Ø OPEN "O", 1, "LIST.EMP"
   2Ø LINE INPUT "NAME? "; N$
   3Ø IF N$="DONE" THEN 6Ø
   4Ø WRITE#1, N$
   5Ø PRINT: GOTO 2Ø
   6Ø CLOSE 1
   RUN
   ```

   Note: The file "LIST.EMP" stores the data you input through the aid of the program, not the program itself (the program manipulates data). To save the program above, you must assign it a name and SAVE it (refer to Chapter 1).

Example

```
SAVE "PAYROLL"
```

would save the program under the name "PAYROLL". This would enable you to edit, add or delete program lines later.

4. To access data in the file, reOPEN it in the "I" (input) mode.

   Example

   ```
   OPEN "I", 1, "LIST.EMP"
   ```

   OPENs the file named LIST.EMP for sequential input, using buffer 1.

5. To read data from the file and assign it to program variables, use either INPUT# or LINE INPUT#.

   Examples

   ```
   INPUT#1, N$
   ```

   reads a string item into N$, using buffer 1 (the buffer used when the file was OPENed).

   ```
   LINE INPUT#1, N$
   ```

   reads an entire line of data into N$, using buffer 1.

   INPUT# and LINE INPUT# each recognize a different set of "delimiters" for reading data from the file. Delimiters are characters that define the beginning or end of a data item. See Chapter 7 for a detailed explanation of these statements.

   Sample Program

   ```
   10  OPEN "I", 1, "LIST.EMP"
   20  IF EOF(1), THEN 100
   30  INPUT#1, N$
   40  PRINT N$
   50  GOTO 20
   100 CLOSE
   ```

Updating a Sequential-Access File

You can add data at the end of a sequential-access file. But if you simply open the file in "O" mode and start writing the data, you will destroy the file's current contents.

Follow these steps instead:

1) OPEN the existing file in the "A" mode for sequential output, after the pointer is positioned at the end-of-data.

2) Write the additional data to the new file.

Now you have a file on disk that includes all the previous data, plus the data you just added in step two.

The following program illustrates this technique. It builds upon the file we previously created under the name LIST.EMP.

Note: Read through the entire program first. If you encounter MBASIC words (commands or functions) that are unfamiliar to you, refer to Chapter 7 for their definitions.

```
NEW
1Ø OPEN "A", 1, "LIST.EMP"
2Ø LINE INPUT "TYPE A NEW NAME OR PRESS <N>"; N$
3Ø IF N$="N" THEN 6Ø
4Ø WRITE#1, N$
5Ø GOTO 2Ø
6Ø CLOSE 1
RUN
```

If you wanted the program to print on your display the information stored in the updated file, add the following lines:

```
7Ø OPEN "I", 1, "LIST/EMP"
8Ø IF EOF(1) THEN 2ØØØ
9Ø LINE INPUT#1, N$
1ØØ PRINT N$
11Ø GOTO 8Ø
2ØØØ CLOSE
RUN
```

Once you have run this program, save it.

Example

```
SAVE "PAYROLL 2" 'saves the new program
```

Direct-Access Files

With a direct-access file, you can access data almost anywhere on disk. It is not necessary to read through all the information, as a with sequential-access file. This is possible because in a direct-access file, information is stored and accessed in distinct units called "records". Each record is numbered.

Creating and accessing direct-access files requires more program steps than sequential-access files. However, direct-access files are more flexible and easier to update. They generally take up less room on disk because MBASIC stores them in packed binary format.

The maximum number of logical records is 32,767. Each record may contain between 1 and 32767 bytes.

The statements and functions used with direct-access files are:

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

These statements and functions are discussed in more detail in Chapters 7.

Creating a Direct-Access File

1. To create the file, OPEN it for direct access in "D" mode.

   Example

   ```
   OPEN, "D", 1, "LISTING", 32
   ```

   opens the file named "LISTING", gives buffer 1 direct access to the file, and sets the record length to 32 bytes. (If the record length is omitted, the default is 256 bytes). Remember that data is passed to and from disk in records.

2. Use the FIELD statement to allocate space in the buffer for the variables that will be written to the file. This is necessary because you must place the entire record into the buffer before putting it into the disk file.

   Example

   ```
   FIELD 1, 2Ø AS N$, 4 AS A$,8 AS P$
   ```

   allocates the first 2Ø positions in buffer 1 to string variable N$, the next four positions to A$, and the next eight positions to P$. N$, A$ and P$ are now "field names". The field statement is ignored after the file is closed.

3. To move data into the buffer, use the LSET statement. Numeric values must be converted 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.

   Example

   ```
   LSET N$=X$
   LSET A$=MKS$(AMT)
   ```

4. To write data from the buffer to a record (within a direct-access disk file), use the PUT statement.

   ```
   PUT 1, CODE%
   ```

   writes the data from buffer 1 to a record with the number CODE%. (The percentage sign at the end of a variable specifies that it will store only integers.)

   The following program writes information to a direct-access file:

   ```
   1Ø OPEN "D", 1, "LISTING", 32
   2Ø FIELD 1, 2Ø AS N$, 4 AS A$, 8 AS P$
   3Ø INPUT "2-DIGIT CODE OR Ø TO END"; CODE%
   35 IF CODE%=Ø THEN 12Ø
   4Ø INPUT "NAME"; X$
   5Ø INPUT "AMOUNT"; AMT
   6Ø INPUT "PHONE"; TEL$
   7Ø LSET N$ = X$
   8Ø LSET A$ = MKS$(AMT)
   ```

```
9Ø LSET P$ = TEL$
1ØØ PUT #1, CODE%
11Ø GOTO 3Ø
12Ø CLOSE
```

Every time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 3Ø becomes the record number.

Once you have run this program, save it.

Example

```
SAVE "DIRECTORY"    'saves the new program
```

Accessing a Direct-Access File

1. OPEN the file in "D" mode ("R" can also be used).

   Example

   ```
   OPEN "D", 1,"FILE",32
   ```

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

   Example

   ```
   FIELD 1, 2Ø AS N$, 4 AS A$, 8 AS P$
   ```

3. Use the GET statement to read the desired record from a direct disk file into a direct buffer.

   Example

   ```
   GET 1, CODE%
   ```

   gets the record numbered CODE% and reads it into buffer 1.

4. Convert string values back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

   Example

   ```
   PRINT N$
   PRINT CVS(A$)
   ```

The program may now access the data in the buffer.

The following program accesses the direct-access file "LISTING" created with the previous program.

```
10 OPEN "D", 1, "LISTING", 32
20 FIELD 1,20 AS N$,4 AS A$,8 AS P$
30 INPUT "2-DIGIT CODE OR 0 TO END"; CODE%
35 IF CODE% = 0 THEN 1000
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##"; CVS(A$)
70 PRINT P$: PRINT
80 GOTO 30
1000 CLOSE
```

After typing this program, SAVE it and RUN it.

Chapter 6/ Introduction To MBASIC Statements And Functions

MBASIC is made up of keywords. These keywords instruct the computer to perform certain operations.

Chapter 7 describes all of MBASIC's keywords. This chapter explains the format used in Chapter 7. It also introduces you to MBASIC's two types of keywords: statements and functions.

Format for Chapter 7

Keyword

Syntax parameter(s) or (argument(s))

explanation of parameters or arguments

Brief definition of keyword.

Detailed definition of keyword.

Example(s)

Sample Program(s)

This format varies slightly, depending on the complexity of each keyword. For instance, some keywords are used alone (without parameters or arguments). Others have several possible syntaxes. As a general rule, definitions for statements are longer than definitions for functions. That is because a statement is a complete instruction to MBASIC, while a function is a built-in subroutine which may only be used as part of a statement.

Some keywords have several sample programs. We added programs to illustrate useful applications which may not be readily apparent. Remember that this manual is to be used as a reference, not a tutorial on how to program in MBASIC.

For a definition of the terms and notation used in Chapter 7, see page 3-5 of the Introduction.

Statements

A program is made up of lines; each line contains one or more statements. A statement tells the computer to perform some operation when that particular line is executed. For example,

```
100 STOP
```

tells the computer to stop executing the program when it reaches line 100.

Statements for assigning values to variables and defining memory space are:

CLEAR	clears all variables, indicates memory to be allocated for program, variable, and stack space.
COMMON	passes variables to a CHAINed program.
DATA	stores data in your program so that you may assign it to a variable.
DEFDBL	defines variables as double precision.
DEF FN	defines a function according to your specifications.
DEFINT	defines variables as integers.
DEFSNG	defines variables as single precision.
DEFSTR	defines variables as strings.
DIM	dimensions an array.
ERASE	erases an array.
LET	assigns a value to a variable (the keyword LET may be omitted).
MID$	replaces a portion of a string.
OPTION BASE	declares the minimum value for array subscripts.
RANDOMIZE	reseeds the random number generator.
READ	reads data stored in the DATA statement and assigns it to a variable.
RESTORE	restores the DATA pointer.
SWAP	exchanges the values of variables.

Statements for altering program sequence:

CHAIN	loads another program and passes variables to the current program.
END	ends a program.
FOR/NEXT	establishes a program loop.
GOSUB	transfers program control to the subroutine
GOTO	transfers program control to the specified line number.
IF...THEN...ELSE	evaluates an expression and performs an operation if conditions are met.
ON...GOSUB	evaluates an expression and branches to a subroutine.
ON...GOTO	evaluates an expression and branches to another program line.
RETURN	returns from a subroutine to the calling program.
STOP	stops program execution.
WHILE...WEND	executes statements in a loop as long as a given condition is true.

Statements for storing and accessing data on disk:

CLOSE	closes access to a disk file.
CVD	restores data from a direct disk file to double precision.
CVI	restores data from a direct disk file to integer.
CVS	restores data from a direct disk file to single precision.
FIELD	organizes a direct-access buffer.
GET	gets a record from a direct-access file.
INPUT#	inputs data from a disk file.
LINE INPUT#	inputs an entire line from a disk file.
LOCK	restricts access to specified portions of a file.
LSET	moves data (and left-justifies it) to a direct file buffer.
MKD$	converts a double-precision number to string for direct-access write.
MKI$	converts an integer to string for direct-access write.
MKS$	converts a single-precision number to string for direct-access write.
OPEN	opens a disk file.
PRINT#	writes data to a sequential disk file.
PRINT# USING	writes data to a disk file using the specified format.

PUT	puts a record into a direct-access file.
RSET	moves data (and right-justifies it) to a direct file buffer.
UNLOCK	releases access restrictions placed on a file.
WRITE#	writes data to a sequential file.

Statements for debugging a program:

CONT	continues program execution.
ERL	returns the line number where an error occurred.
ERR	returns an error code after an error.
ERROR	simulates the specified error.
ON ERROR GOTO	sets up an error-trapping routine.
RESUME	terminates an error-handling routine.
TROFF	turns the tracer off.
TRON	turns the tracer on.

Statements for inputting data from the keyboard or outputting data to the video display or the line printer:

CLS	clears the display.
INPUT	inputs data from the keyboard.
LINE INPUT	inputs an entire line from the keyboard.
LIST	lists a program to the display.
LLIST	lists program to line printer.
LPRINT	prints an entire line to the display.
LPRINT USING	prints to printer using a specified format.
PRINT	prints to the display.
PRINT USING	prints to the display using a specified format.
PRINT @	specifies exactly where printing is to begin.
PRINT TAB	moves cursor to specified tab position.
WIDTH	sets the printed line width for the file which is opened for output.
WRITE	prints data at the display.

Statements for performing system functions or entering other modes of operation:

AUTO	automatically numbers program lines.
CALL	calls an assembly-language subroutine.
DELETE	erases program lines from memory.
EDIT	edits program lines.
KILL	deletes a disk file.
LOAD	loads a program from disk.
MERGE	merges a disk program with a resident program.
NAME	renames a disk file.
NEW	erases a program from RAM.
POKE	writes a byte into a memory location.

RENUM	renumbers a program.
RUN	executes a program.
SAVE	saves a program on disk.
SHELL	calls a TRS-XENIX command and returns to MBASIC.
SYSTEM	returns to TRS-XENIX.

Functions

A function is a built-in subroutine. It may only be used as part of a statement.

Most MBASIC functions return numeric or string data by performing certain built-in routines. Special print functions are used to control the video display.

Numeric Functions (return a number):

ABS	computes the absolute value.
ASC	returns the ASCII code.
ATN	computes the arctangent
CDBL	converts to double precision.
CINT	returns the largest integer not greater than the parameter.
COS	computes the cosine.
CSNG	converts to single precision
EXP	computes the natural exponential.
FIX	truncates to whole number.
FRE	returns the number of bytes in memory not being used.
INSTR	searches for a specified string.
INT	returns the largest whole number not greater than the argument.
LEN	returns the length of the string.
LOG	computes the natural logarithm.
MEM	returns the amount of memory.
PEEK	returns a byte from a memory location.
RND	returns the pseudorandom number.
SGN	returns the sign.
SHELL	executes a TRS-XENIX command, returns the process I.D., and returns to MBASIC.
SIN	calculates the sine.
SQR	calculates the square root.
TAN	computes the tangent.
VAL	evaluates a string.
VARPTR	returns the address for a buffer.

String Functions (return a string value):

CHR$	returns the specified character
DATE$	returns today's date.
HEX$	converts a decimal value to a hexadecimal string.
LEFT$	returns the left portion of a string.
MID$	returns the mid-portion of a string.
OCT$	converts a decimal value to an octal string.
RIGHT$	returns the right portion of a string.
SPACE$	returns a string of spaces.
STR$	converts to string type.
STRING$	returns a string of characters.
TIME$	returns the time.

Input/Output Functions (perform miscellaneous functions with the keyboard, display, disk files, or "devices"):

INKEY$	returns the keyboard character
INPUT$	returns a string of characters from either the keyboard or a sequential disk file.
POS	returns the cursor column position on the display.
SPC	prints spaces to the display.
EOF	checks for end-of-file.
LOC	returns the current disk file record number.
LOF	returns the disk file's end-of-file record number.
LPOS	returns the column position at the line printer.
TAB	spaces to a tab position.

ABS — Function

ABS(number)

Computes the absolute value of number.

ABS returns the absolute value of the argument, that is, the magnitude of the number without respect to its sign.

If number is greater than or equal to zero, ABS(number)=number. If number is less than zero, ABS(negative number)=number.

Example

```
X = ABS(Y)
```

computes the absolute value of Y and assigns it to X.

Sample Program

```
100 INPUT "WHAT'S THE TEMPERATURE OUTSIDE (DEGREES F)"; TEMP
110 IF TEMP < 0 THEN PRINT "THAT'S" ABS(TEMP) "BELOW ZERO!
    BRR!": END
120 IF TEMP = 0 THEN PRINT "ZERO DEGREES! MITE COLD!": END
130 PRINT TEMP "DEGREES ABOVE ZERO?  BALMY!": END
```

KEYWORDS

ASC — Function

ASC(string)

Returns the ASCII code for the first character of string .

The value is returned as a decimal number. If string is null, an "Illegal function call" error occurs.

Example

```
PRINT ASC("A")
```

prints 65, the ASCII code for "A".

Sample Program

ASC can be used to make sure that a program is receiving the proper input. Suppose you've written a program that requires the user to input hexadecimal digits Ø-9, A-F. To make sure that only those characters are input, and exclude all other characters, you can insert the following routine.

```
1ØØ INPUT "ENTER A HEXADECIMAL VALUE (Ø-9,A-F)";N$
11Ø A=ASC(N$)                'get ASCII code
12Ø IF A>47 AND A<58 OR A>64 AND A<71 THEN PRINT "OK.": GOTO
    1ØØ
13Ø PRINT "VALUE NOT OK." : GOTO 1ØØ
```

ATN Function

ATN(number)

Computes the arctangent of number in radians.

ATN returns the angle whose tangent is number. The result is always double precision, regardless of number's numeric type.

To convert this value to degrees, multiply ATN(number) by 57.29578.

Example

```
X = ATN(Y/3)
```

computes the arctangent of Y/3 and assigns the value to X.

AUTO Statement

AUTO [line][,increment]

Automatically generates a line every time you press <ENTER>.

AUTO begins numbering at line and displays the next line using increment. The default for both values is 10. A period (.) can be substituted for line. In this case, MBASIC uses the current line number.

IF AUTO generates a line number that has already been used, it displays an asterisk after the number. To save the existing line, press <ENTER> immediately after the asterisk. AUTO then generates the next line number.

To turn off AUTO, press <CTRL><C> or <BREAK>. The current line is canceled and MBASIC returns to command level.

Examples

```
AUTO
```

generates lines 10, 20, 30, 40....

```
AUTO 100, 50
```

generates lines 100, 150, 200, 250....

CALL Statement

CALL variable [paremeter list]

Transfers program control to a machine language subroutine stored at variable.

Variable contains the address where the subroutine starts in memory. Variable may not be an array variable.

If variable is zero, then the character string "<variable>" is used to search for the location to start execution. "<variable>" has to match the symbolic subroutine name used when the subroutine was either assembled or compiled. Remember that all variable names must be in capitals. The C compiler modifies all subrountine names by prefixing an underscore. Thus, the subroutine _FOO is represented internally as FOO. MBASIC first searches for FOO. If this is not found, then it searches for _FOO and starts execution at _FOO. The value of the address (in this case FOO) is returned in <variable>. Subsequent calls using the same <variable> avoids the search phase and starts execution at the variable's value. For this reason, all <variable's> used must be either single or double precision. They cannot be an integer; an integer, is not large enough to hold the 24-bit address of the 68ØØØ.

Example

```
12Ø CALL MYROUT(I,J,K)
```

Parameter list contains the values that are passed to the external subroutine. Parameter list may contain only variables.

See Appendix E.

Example

```
11Ø MYROUT = &HDØØØ
12Ø CALL MYROUT(I,J,K)
```

transfers program control to an assembly-language routine stored at address DØØØ. The values of I, J. and K are passed to the routine.

CDBL Function

CDBL(number)

Converts number to double precision.

CDBL returns a 14-digit value. This function may be useful if you want to force an operation to be performed in double precision, even though the operands are single precision or integers.

Sample Program

```
21Ø A=454.67
22Ø PRINT A; CDBL(A)
RUN
 454.67 454.67
Ok
```

CHAIN Statement

CHAIN [MERGE] filespec [,line] [,ALL] [,DELETE line-line]

Loads a MBASIC program named filespec, chains it to a "main" program, and begins running it.

Line is the first line to be run in the CHAINed program. If omitted, execution begins at the first program line of the CHAINed program. Line is not affected by the RENUM command.

The ALL option passes every variable in the main program to the CHAINed program. If omitted, the main program must contain a COMMON statement to pass variables.

The MERGE option "overlays" the lines of filespec with the main program. See MERGE to understand how MBASIC overlays (merges) program lines.

The DELETE option deletes lines in the overlay so that you can MERGE in a new overlay.

Examples

```
CHAIN PROG2
```

loads PROG2, chains it to the main program currently in memory, and begins executing it.

```
CHAIN SUBPROG.BAS, ALL
```

loads, chains and executes SUBPROG.BAS. The values of all the variables in the main program are passed to SUBPROG.BAS.

Sample Program 1

```
1Ø REM THIS PROGRAM DEMONSTRATES CHAINING USING COMMON TO PASS
   VARIABLES.
2Ø DIM A$(2),B$(2)
3Ø COMMON A$(),B$()
4Ø A$(1)="VARIABLES IN COMMON MUST BE ASSIGNED"
5Ø A$(2)="VALUES BEFORE CHAINING"
```

```
60 B$(1)="":B$(2)=""
70 CHAIN"PROG2"
80 PRINT: PRINT B$(1): PRINT: PRINT B$(2): PRINT
90 END
```

Save this program using the 'A' option as "PROG1". Save "filespec", A saves the program in ASCII format. Enter the NEW command then type:

```
10 REM THE STATEMENT "DIM A$(2),B$(2)" MAY ONLY BE EXECUTED
   ONCE.
20 REM HENCE, IT DOES NOT APPEAR IN THIS MODULE.
30 COMMON A$(),B$()
40 PRINT: PRINT A$(1);A$(2)
50 B$(1)="NOTE HOW THE OPTION OF SPECIFYING A STARTING LINE
   NUMBER"
60 B$(2)="WHEN CHAINING AVOIDS THE DIMENSION STATEMENT IN
   'PROG1'."
70 CHAIN "PROG1",80
80 END
```

Save this program as "PROG2" using the 'A' option. Type NEW. Load "PROG1" and run it. Your screen should display:

```
VARIABLES IN COMMON MUST BE ASSIGNED VALUES BEFORE
CHAINING.  NOTE HOW THE OPTION OF SPECIFYING A STARTING
LINE NUMBER WHEN CHAINING AVOIDS THE DIMENSION
STATEMENT IN "PROG1".
```

Sample Program 2

Type NEW and this program:

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING USING THE MERGE AND
   ALL OPTIONS.
20 A$="MAINPRG"
30 CHAIN MERGE "OVRLAY1",1000,ALL
40 END
```

Save this program as "MAINPROG". Enter the NEW command then type:

```
1000 PRINT A$;"HAS CHAINED TO OVRLAY1."
1010 A$="OVRLAY1"
1020 B$="OVRLAY2"
1030 CHAIN MERGE "OVRLAY2",1000,ALL,DELETE 1020-1050
1040 END
```

Save this program on disk as "OVRLAY1" using the A option. Enter the NEW command then type:

```
1000 PRINT A$; " HAS CHAINED TO ";B$;"."
1010 END
```

Save this program on disk as "OVRLAY2" using the A option. Load "MAINPROG" and run it.

Your screen will display:

```
MAINPROG HAS CHAINED TO OVRLAY1.
OVRLAY1 HAS CHAINED TO OVRLAY2.
```

Note:

The CHAIN statement leaves the files OPEN and preserves the current OPTION BASE setting.

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

When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

CHR$ Function

CHR$(code)

Returns the character corresponding to an ASCII or control code.

This is the inverse of the ASC function. CHR$ is commonly used to send a special character to the display.

Examples

```
PRINT CHR$(35)
```

prints the character corresponding to ASCII code 35 (the character is #).

Sample Program

The following program lets you investigate the effect of printing each of the 256(Ø-255) codes on the display.

```
1ØØ CLS
11Ø INPUT"TYPE IN THE CODE (Ø-255)";C
12Ø PRINT CHR$(C); "    JUST PRINTED CODE" C
13Ø GOTO 11Ø
```

CINT Function

CINT(number)

Converts number to integer representation.

CINT rounds the fractional portion of number to make it an integer.

For example, CINT(1.5) returns 2; CINT(-1.5) returns -2. The result is a two-byte integer.

Sample Program

```
PRINT CINT(17.65)
 18
Ok
```

CLEAR — Statement

CLEAR

CLEAR ,[memory space] ,[stack space]

Clears the value of all variables, closes all open files and sets the amount of memory space and stack space.

Memory space must be an integer. It indicates how many bytes are to be allocated for program, variable, and stack space. The default is the current amount allocated. 31,Ø85 bytes are allocated upon entry to MBASIC. Memory space may be as large as 16,777,215. Memory space is position dependent; if omitted, the commas must still be specified.

Stack space must also be an integer. This sets aside memory for temporarily storing internal data and addresses during subroutine calls. This number may be as large as 16,777,215. The default is the current amount allocated (1,Ø24 bytes are allocated upon entry to MBASIC). An "Out of memory" error occurs if there is insufficient stack space for program execution. The FRE and MEM functions do not count stack space.

Note: MBASIC allocates string space dynamically. An "Out of string space" error occurs only if no free memory is left for MBASIC.

Since CLEAR initializes all variables, you must use it near the beginning of your program, before any variables have been defined and before any DEF statements. CLEAR used with either of its two options causes data written using LPRINT statements to be output to the printer.

Examples:

```
CLEAR
```

clears all variables and closes all files. Memory allocations remain unchanged.

```
CLEAR ,32ØØ
```

clears all variables and closes all files; allocates 32ØØ bytes for program, variables, and stack space.

```
CLEAR ,61000, 200
```

clears all variables and closes all files; allocates 61000 bytes for program and variable space, and 200 bytes for stack space.

```
CLEAR ,,2000
```

clears all variables and closes all files; 2000 bytes of memory are allocated for stack space.

CLOSE Statement

CLOSE [buffer,...]

Closes access to a file.

Buffer is the number of the buffer used to OPEN the file. If no buffers are specified, all open files are closed.

This command terminates access to a file through the specified buffers. If a buffer was not assigned in a previous OPEN statement, then

CLOSE buffer

has no effect.

Do not remove a diskette which contains an Open file. CLOSE the file first. This is because the last records may not have been written to disk yet. Closing the file writes the data, if it hasn't already been written. Also, the disk must be unmounted before it can be removed from the drive. Please refer to the /etc/mount and /etc/umount commands in the XENIX User's Guide.

See also OPEN and the chapter on "Disk Files".

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%.

CLS Statement

CLS

Clears the screen.

CLS fills the display with blanks and moves the cursor to the upper-left corner. Alphanumeric characters and graphics blocks are wiped out.

Sample Program

```
54Ø CLS
55Ø FOR I = 1 TO 24
56Ø PRINT STRING$(79,33)
57Ø NEXT I
58Ø GOTO 54Ø
```

COMMON Statement

COMMON variable,...

Passes one or more variable(s) to a CHAINed program.

This statement is used in conjunction with CHAIN. COMMON may appear anywhere in a program, but we recommend using it at the beginning.

The same variable cannot appear in more than one COMMON statement. To specify array variables, append "( )" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Note: Array variables used in a COMMON statement must have been declared in a DIM statement.

Example

```
 99 DIM D()
1ØØ COMMON A, B, C, D(),G$
11Ø CHAIN "PROG3", 1Ø
```

line 1ØØ passes variables A, B, C, D and G$ to the CHAIN command in line 11Ø.

Note: The CLEAR command clears COMMON variables.

See also CHAIN and CLEAR.

CONT Statement

CONT

Resumes program execution.

You may only use CONT if the program was stopped by the <BREAK> key, <CTRL><C>, a STOP or an END statement in the program.

CONT is primarily a debugging tool. During a break or stop in execution, you may examine variable values (using PRINT) or change these values using immediate lines or commands. Then type CONT <ENTER> and execution continues with the current variable values.

You cannot use CONT after EDITing your program lines or otherwise changing your program. CONT is also invalid after execution has ended normally.

Example

```
10 INPUT A, B, C
20 K=A^2405.3
30 L=B^3/.26
40 STOP
50 M=C*40*K+100: PRINT M
```

Run this program.
You will be prompted with

```
?
```

Type: 1, 2, 3

The program will halt execution and then you can type any immediate command that you wish.

For example:

```
PRINT L
```

displays 30.7692307692 3. You can also change the value of A, B, or C.

For example:

```
C=4
```

changes the value of C in the program. Type:

```
CONT
```

Your screen will display 259.99999996152

See also STOP.

COS — Function

COS(number)

Computes the cosine of number.

COS returns the cosine of number in radians. The angle must be given in radians. When number is in degrees, use COS(number * .Ø1745329).

The result is always double precision.

Examples

```
Y = COS(X * .Ø1745329)
```

returns the cosine of X, if X is an angle in degrees.

```
PRINT COS(5.8) - COS(85 * .42)
```

prints the arithmetic (not trigonometric) difference of the two cosines.

CSNG Function

CSNG(number)

Converts number to single precision.

Example

```
PRINT CSNG(.1453885Ø9)
```

prints .145389. Note that only six significant digits are printed and the sixth digit (9) is rounded.

Sample Program

```
28Ø V# = 876.2345678#
29Ø PRINT V#; CSNG(V#)
RUN
 876.2345678    876.235
```

CVD, CVI, CVS — Functions

CVD(eight-byte string)
CVS(four-byte string)
CVI(two-byte string)

Convert string values to numeric values.

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 direct-access file buffer.

CVD converts an eight-byte string to a double-precision number. CVS converts a four-byte string to a single-precision number. CVI converts a two-byte string to an integer.

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

Examples

Suppose the name GROSSPAY$ references an eight-byte field in a direct-access file buffer, and after GETing a record, GROSSPAY$ contains an MKD$ representation of the number 13123.38. Then the statement

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

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

Sample Program

```
142Ø OPEN "R", 1, "TEST.DAT"
143Ø FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
144Ø GET 1
145Ø PRINT CVI(I1$), CVS(I2$), CVD(I3$)
146Ø CLOSE
```

This program opens a file named TEST.DAT which is assumed to have been previously created. (For the program which creates the file, see the section on MKD$, MKI$, and MKS$). CVI, CVS, and CVD are used to convert string data back to numeric form.

DATA Statement

DATA constant,...

Stores numeric and string constants to be accessed by a READ statement.

This statement may contain as many constants (separated by commas) as will fit on a line. Each will be read sequentially, starting with the first constant in the first DATA statement, and ending with the last item in the last DATA statement.

Numeric expressions are not allowed in a DATA list. If your string values include leading blanks, colons, or commas, you must enclose these values in double quotation marks.

DATA statements may appear anywhere it is convenient in a program. The data types in a DATA statement must match up with the variable types in the corresponding READ statement, otherwise a syntax error in line # occurs.

Examples

```
1340 DATA NEW YORK, CHICAGO, LOS ANGELES, PHILADELPHIA, DETROIT
```

stores five string data items. Note that quote marks aren't needed, since the strings contain no delimiters and the leading blanks are not significant.

```
1350 DATA 2.72, 3.14159, 0.0174533, 57.29578
```

stores four numeric data items.

```
1360 DATA "SMITH, T.H.", 38, "THORN, J.R.", 41
```

stores both types of constants. Quote marks are required around the first and third items.

Sample Program

```
LIST
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z
30 DATA DENVER, COLORADO, 80211
```

```
40 PRINT C$,S$,Z
```

This program READS string and numeric data from the DATA statement in line 30, and displays on your screen:

```
CITY        STATE       ZIP
DENVER      COLORADO     80211
```

DATE$ Function

DATE$

Returns the current date in the specified string variable.

The operator sets the date when TRS-XENIX is started up.

During a program, if you request the date, MBASIC assigns it to the specified string variable. For example:

```
A$ = DATE$
```

assigns the date string to A$. If you print A$:

```
PRINT A$
```

it is displayed in this fashion:

```
Ø3-12-1983
```

which means March 12, 1983.

Sample Program

```
1ØØØ PRINT "Inventory Check:"
11ØØ IF DATE$ = Ø1-31-198Ø THEN PRINT "Today is the last day of
     January 198Ø. Time to perform monthly inventory.": END
112Ø A$ = LEFT$ (DATE$,5): B$ = RIGHT$ (A$,2)
113Ø B = VAL(B$)
114Ø PRINT 31-B "days until inventory time"
```

DEFDBL/INT/SNG/STR Statement

DEFDBL letter,...
DEFINT letter,...
DEFSNG letter,...
DEFSTR letter,...

Defines any variables beginning with letter(s) as: (DBL) double precision, (INT) integer, (SNG) single precision, or (STR) string.

Note: A type declaration character always takes precedence over a DEF statement.

Examples

```
10 DEFDBL L-P
```

classifies all variables beginning with the letters L through P as double-precision variables. Their values include 14 digits of precision, and all 14 are printed out.

```
10 DEFSTR A
```

classifies all variables beginning with the letter A as string variables.

```
10 DEFINT I-N, W,Z
```

classifies all variables beginning with the letters I through N along with W and Z as integer variables. Their values are in the range -32768 to 32767.

```
10 DEFSNG I, Q-T
```

classifies all variables beginning with the letters I and Q through T as single-precision variables. Their values include seven digits of precision, and all seven are printed out.

DEF FN Statement

DEF FN function name [(argument,...)] =function definition

Defines function name according to your function definition.

Function name must be a valid variable name. The type of variable used determines the type of value the function will return. For example, if you use a single-precision variable, the function will always return single-precision values.

Argument represents those variables in function definition that are to be replaced when the function is called. If you enter several variables, separate them by commas.

Function definition is an expression that performs the operation of the function. A variable used in a function definition may or may not appear in argument. If it does, MBASIC uses the argument's to perform the function. Otherwise, it uses the current value of the variable.

Once you define and name a function (by using this statement), you can call it and MBASIC performs the associated operations.

Examples

```
DEF FNR=RND(9Ø)+9
```

defines a function FNR to return a random value between 1Ø and 99. Notice that the function can be defined with no arguments.

```
21Ø DEF FNW#(A#,B#)=(A#-B#)*(A#-B#)
28Ø T=FNW#(I#,J#)
```

defines function FNW# in line 21Ø. Line 28Ø calls that function and replaces parameters A# and B# with parameters I# and J#. (We assume that I# and J# were assigned values elsewhere in the program).

Note: Using a variable as an parameter in a DEF FN statement has no effect on the value of that variable. You may use that variable in another part of the program without interference from DEF FN.

DELETE Statement

DELETE line1 - line2

Deletes from line1 through line2 of a program in memory.

A period (".") can be substituted for either line1 or line2 to indicate the current line number.

Examples

DELETE 7Ø

deletes line 7Ø from memory. If there is no line 7Ø, an error will occur.

DELETE 5Ø-11Ø

deletes lines 5Ø through 11Ø inclusive.

DELETE -4Ø

deletes all program lines up to and including line 4Ø.

DELETE -.

deletes all program lines up to and including the line that has just been entered or edited.

DELETE .

deletes the program line that has just been entered or edited.

DIM Statement

DIM array (dimension(s)), array (dimension(s)),...

Sets aside storage for arrays with the dimensions you specify.

This statement is useful for structured data processing.

Arrays may be of any type: string, integer, single precision or double precision, depending on the type of variable used to name the array. If no type is specified, the array is classified as single precision.

When you create the array, MBASIC reserves space in memory for each element of the array. All elements in a newly- created array are set to zero (numeric arrays) or the null string (string arrays).

Note: The lowest element in a dimension is always zero, unless OPTION BASE 1 has been used. See OPTION BASE for more information.

Arrays can be created implicitly, without explicit DIM statements. Simply refer to the desired array in a MBASIC statement, e.g.,

```
A(5) = 3ØØ
```

creates array A and assigns element A(5) the value of 3ØØ. Each dimension of an implicitly-defined array is 11 elements deep, subscripts Ø-1Ø.

Examples

```
DIM AR(1ØØ)
```

sets up a one-dimensional array AR( ), containing 1Ø1 elements: AR(Ø), AR(1), AR(2),..., AR(98), AR(99), and AR(1ØØ).

Note: The array AR( ) is completely independent of the variables AR.

```
DIM L1%(8,25)
```

sets up a two-dimensional array L1%(,), containing 9 x 26 integer elements, L1%(Ø,Ø), L1%(1,Ø), L1%(2,Ø),...,L1%(8,Ø), L1%(Ø,1), L1%(1,1),...,L1%(8,1),...,L1%(Ø,25), L1%(1,25),..., L1%(8,25).

Two-dimensional arrays like AR(,) can be thought of as a table in which the first subscript specifies a row position, and the second subscript specifies a column position:

```
Ø,Ø   Ø,1   Ø,2   Ø,3   ...   Ø,23   Ø,24   Ø,25
1,Ø   1,1   1,2   1,3   ...   1,23   1,24   1,25
.
.
.
7,Ø   7,1   7,2   7,3   ...   7,23   7,24   7,25
8,Ø   8,1   8,2   8,3   ...   8,23   8,24   8,25
```

```
DIM B1(2,5,8),  CR(2,5,8),  LY$(5Ø,2)
```

sets up three arrays:
B1(,,) and CR (, ,) are three-dimensional, each containing 3*6*9 elements.
LY(,) is two-dimensional, containing 51*3 string elements.

EDIT Statement

EDIT line

Enters the edit mode so that you can edit line.

See the chapter on the "Edit Mode" for more information.

Examples

```
EDIT  1ØØ
```

enters edit mode at line 1ØØ.

```
EDIT .
```

enters edit mode at current line.

END Statement

END

Ends execution of a program.

This statement may be placed anywhere in the program. It forces execution to end at some point other than the last sequential line. When the END statement is encountered it spools the LPRINT buffer to the line printer.

An END statement at the end of a program is optional.

Sample Program

```
4Ø INPUT S1, S2
5Ø GOSUB 1ØØ
55 PRINT H
6Ø END
1ØØ H=SQR(S1*S1 + S2*S2)
11Ø RETURN
```

line 6Ø prevents program control from "crashing" into the subroutine. Line 1ØØ may only be accessed by a branching statement, such as GOSUB in line 5Ø.

EOF Function

EOF(buffer)

Detects the end of a file.

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.

EOF(buffer) returns Ø (false) when the EOF record has not been read yet, and -1 (true) when it has been read. The buffer number must access an OPEN file.

This function is valid for any file which is opened for input. A file opened to "KYBD: " is at its end when CTL-D is pressed.

Sample Program

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 15ØØ "passes", so the program branches out of the disk access loop.

```
147Ø DIM A(1ØØ)          'ASSUMING THIS IS A SAFE VALUE
148Ø OPEN "I", 1, "DATA.TXT"
149Ø I% = Ø
15ØØ IF EOF(1) THEN GOTO 154Ø
151Ø INPUT#1, A(I%)
152Ø I% = I% + 1
153Ø GOTO 15ØØ
154Ø REM   PROG.  CONT.  HERE AFTER DISK INPUT
```

ERASE Statement

ERASE array,...

Erases one or more arrays from a program.

This lets you to either redimension arrays, or use their previously allocated space in memory for other purposes.

If one of the parameters of ERASE is a variable name which is not used in the program, an "Illegal Function Call" occurs.

Example

```
44Ø DIM C(1Ø), F(1Ø)
45Ø ERASE C,F
46Ø DIM F(99)
```

line 45Ø erases arrays C and F. Line 46Ø redimensions array F.

ERL Statement

ERL

Returns the line number in which the most recent error occurred.

This function is primarily used inside an error-handling routine. If no error has occurred when ERL is called, line number Ø is returned. Otherwise, ERL returns the line number in which the error occurred. If the error occurred in the command mode, 65535 (the largest number representable in two bytes) is returned.

Examples

```
PRINT ERL
```

prints the line number of the error.

```
E = ERL
```

stores the error's line number for future use.

?erl - prints line # on which error occured

ERR Statement

ERR

Returns the error code (if an error has occurred).

ERR is only meaningful inside an error-handling routine accessed by ON ERROR GOTO. See Appendix A for MBASIC's Error Codes.

Example

```
IF ERR = 7 THEN 1000 ELSE 2000
```

branches the program to line 1000 if the error is an "Out of Memory" error (code 7); if it is any other error, control goes instead to line 2000.

?err — will print error code

ERROR Statement

ERROR code

Simulates a specified error during program execution.

Code is an integer expression in the range Ø to 255 specifying one of MBASIC's error codes.

This statement is mainly used for testing an ON ERROR GOTO routine. When the computer encounters an ERROR code statement, it proceeds as if the error corresponding to that code had occurred. (Refer to the Appendices for a listing of error codes and their meanings).

Example

```
ERROR 1
```

a "Next Without For" error (code 1) "occurs" when MBASIC reaches this line.

Sample Program

```
1Ø ON ERROR GOTO 7Ø
2Ø READ A
3Ø PRINT A
4Ø GOTO 2Ø
5Ø PRINT "DATA HAS BEEN READ IN"
6Ø END
7Ø IF ERR = 4 THEN RESUME 5Ø
8Ø ON ERROR GOTO Ø
9Ø DATA 4, -2, Ø, 5, 9, 2, 2, 31, 7, 13, 1
```

This program "traps" the Out of Data error, since 4 is the code for that error.

EXP Function

EXP(number)

Calculates the natural exponential of number.

Returns e (base of natural logarithms) to the power of number. This is the inverse of the LOG function; therefore, number = LOG(EXP(number)). The number you supply must be less than or equal to 87.3365.

The result is always double precision.

Example

```
PRINT EXP(-2)
```

prints the exponential value .13533528323662.

Sample Program

```
31Ø INPUT "NUMBER"; N
32Ø PRINT "E RAISED TO THE N POWER IS" EXP(N)
```

FIELD Statement

FIELD buffer, length AS field name,...

Divides a direct-access buffer into one or more fields. Each field is identified by field name and is the length you specify.

Field name must be a string variable.

This divides a direct file buffer so that you can send data from memory to disk and disk to memory. FIELD must be run prior to GET or PUT.

Before "fielding" a buffer, use an OPEN statement to assign that buffer to a particular disk file. (The random-access mode, i.e., OPEN "R",... must be used). The sum of all field lengths can be less than or equal the record length assigned when the file was OPENed.

You may use the FIELD statement any number of times to "re-field" a file buffer. "Fielding" a buffer does not clear the buffer's contents; only the means of accessing it. Field definitions are ignored after a file is closed.

See also the chapter on "Disk Files", OPEN, CLOSE, PUT, GET, LSET, and RSET.

Example

```
FIELD 3, 128 AS A$, 128 AS B$
```

tells MBASIC to assign two 128-byte fields to the variables A$ and B$. If you now print A$ or B$, you will see the contents of the field. Of course, this value would be meaningless unless you have previously used GET to read a 256-byte record from disk.

Note 1: 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, and MKD$/CVD) for converting numbers to strings and strings to numbers.

```
FIELD 3, 16 AS NM$, 25 AS AD$, 1Ø AS CY$, 2 AS ST$,7 AS ZP$
```

assigns the first 16 bytes of buffer 3 to field NM$; the next 25 bytes to AD$; the next 1Ø to CY$; the next 2 to ST$; and the next 7 to ZP$.

Note 2: String variables assigned in the FIELD statement should only be used with LSET and RESET. **DO NOT** redefine string variables used in FIELD statements within your program.

FIX Function

FIX(number)

Returns the truncated integer of number.

All digits to the right of the decimal point are simply chopped off, so the resultant value is a whole number.

The result is the same precision as the argument (except for the fractional portion).

Examples

```
PRINT FIX (2.6)
```

prints 2.

```
PRINT FIX(-2.6)
```

prints -2.

FOR/NEXT Statement

FOR variable = initial value **TO** final value **[STEP** increment**]**

NEXT [variable**]**

Establishes a program loop.

A loop allows for a series of program statements to be executed over and over a specified number of times.

MBASIC executes the program lines following the FOR statement until it encounters a NEXT. At this point, it increases variable by STEP increment. If the value of variable is less than or equal to final value, MBASIC branches back to the line after FOR, and repeats the process. If variable is greater than final value, it completes the loop and continues with the statement after NEXT.

If increment has a negative value, then the final value of variable is actually lower than the initial value. MBASIC always sets the final value for the loop variable before setting the initial value.

Note: MBASIC skips the body of the loop if initial value times the sign of STEP increment exceeds final value times the sign of STEP increment.

Example

```
20 FOR H=1 TO 2 STEP -2
30 PRINT H
40 NEXT H
```

the initial value of H times the sign of STEP increment is greater than the final value of H times the sign of STEP increment, therefore MBASIC skips the body of the loop. (The sign of STEP increment is negative in this case).

Sample Program

```
820 I=5
830 FOR I = 1 TO I + 5
840 PRINT I;
850 NEXT
RUN
```

this loop is executed ten times. It produces the following output:

```
1  2  3  4  5  6  7  8  9  1Ø
```

Nested Loops

FOR/NEXT loops may be "nested". That is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. The NEXT statement for the inside loop must appear before the NEXT for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

Sample Program

```
88Ø FOR I = 1 TO 3
89Ø PRINT "OUTER LOOP"
9ØØ FOR J = 1 TO 2
91Ø PRINT "INNER LOOP"
92Ø NEXT J
93Ø NEXT I
```

This program performs three "outer loops" and within each, two "inner loops".

NEXT can be used to close nested loops by listing the counter-variables. For example, delete line 92Ø and change 93Ø to:

```
NEXT J, I
```

Note: In nested loops, if the variable(s) in the NEXT statement is omitted, the NEXT statement matches the most recent FOR statement.

FRE Function

FRE(dummy number)

Returns the amount of free memory space.

FRE returns the amount of space in bytes.

Examples

```
PRINT FRE(Ø)
```

prints the amount of free memory. The memory that the program stack occupies is included in this total.

GET Statement

GET buffer [,record]

Gets a record from a direct disk file and places it in a buffer.

Before using GET, you must OPEN the file and assign it a buffer.

When MBASIC encounters GET, it reads the record number from the file and places it into the buffer. The actual number of bytes read equals the record length set when the file is OPENed.

If record is omitted, MBASIC gets the next record (after the last GET) and reads it into the buffer.

Examples

```
GET 1
```

gets the next record into buffer 1.

```
GET 1, 25
```

gets record 25 into buffer 1.

GOSUB Statement

GOSUB line

Goes to a subroutine, beginning at line.

You can call a subroutine as many times as you want. When the computer encounters RETURN in the subroutine, it returns control to the statement which follows GOSUB.

GOSUB is similar to GOTO in that it may be preceded by a test statement. Every subroutine must end with a RETURN.

Example

```
GOSUB 1000
```

branches control to the subroutine at 1000.

Sample Program

```
260 GOSUB 280
270 PRINT "BACK FROM SUBROUTINE": END
280 PRINT "EXECUTING THE SUBROUTINE"
290 RETURN
```

transfers control from line 260 to the subroutine beginning at line 280. Line 290 instructs the computer to return to the statement immediately following GOSUB.

GOTO Statement

GOTO line

Goes to the specified line.

When used alone, GOTO line results in an unconditional (automatic) branch. However, test statements may precede the GOTO to effect a conditional branch.

You can use GOTO in the command mode as an alternative to RUN. This lets you pass values assigned in the command mode to variables in the execute mode.

Example

```
GOTO 1ØØ
```

transfers control automatically to line 1ØØ.

Sample Program

```
1Ø READ R
2Ø PRINT "R=";R
3Ø A=3.14*R^2
4Ø PRINT "AREA =";A
5Ø GOTO 1Ø
6Ø DATA 5,7,12
RUN
```

line 1Ø reads each of the data items in line 6Ø; line 5Ø returns program control to line 1Ø. This enables MBASIC to calculate the area for each of the data items.

HEX$ Function

HEX$(number)

Calculates the hexadecimal value of number.

HEX$ returns a string which represents the hexadecimal value of the argument. The value returned is like any other string: it cannot be used in a numeric expression.That is, you cannot add hex strings. You can concatenate them, though.

Examples

```
PRINT HEX$(3Ø), HEX$(5Ø), HEX$(9Ø)
```

prints the following strings:

```
1E        32        5A
```

```
Y$ = HEX$(X/16)
```

Y$ is the hexadecimal string representing the integer quotient X/16.

```
1Ø Y$ = HEX$(15) + HEX$(15)
2Ø PRINT Y$
```

prints FF.

IF...THEN...ELSE — Statement

IF expression **THEN** statement(s) **or** line [**ELSE** statement(s)] **or** [line]

Tests a conditional expression and makes a decision regarding program flow.

If expression is true, control proceeds to the THEN statement or line. If not, control jumps to the matching ELSE statement, line, or down to the next program line.

Examples

```
IF X > 127 THEN PRINT "OUT OF RANGE" :  END
```

passes control to PRINT, then to END if X is greater than 127. If X is not greater than 127, control jumps down to the next line in the program, skipping the PRINT and END statements.

```
If A < B PRINT "A < B" ELSE PRINT "B < A"
```

tests the first expression, if true, prints "A < B". Otherwise, the program jumps to the ELSE statement and prints "B < A".

```
IF X > Ø AND Y <> Ø THEN Y = X + 18Ø
```

assigns the value X + 18Ø to Y if both expressions are true. Otherwise, control passes directly to the next program line, skipping the THEN clause.

```
IF A$ = "YES" THEN 21Ø ELSE IF A$ = "NO" THEN 4ØØ ELSE 37Ø
```

branches to line 21Ø if A$ is YES. If not, the program skips over to the first ELSE, which introduces a new test. If A$ is NO, then the program branches to line 4ØØ. If A$ is any value besides NO or YES, the program branches to line 37Ø.

IF THEN ELSE statements may be nested. However, you must take care to match up the IFs and ELSEs. (If the statement does not contain the same number of ELSE's and IF's, each ELSE is matched with the closest unmatched IF.)

Sample Program

```
1Ø4Ø INPUT "ENTER TWO NUMBERS"; A, B
1Ø5Ø IF A > B THEN PRINT "A > B"; ELSE IF A < B THEN PRINT
     "A < B"; ELSE PRINT "A = B"
```

This program prints the relationship between the two numbers entered.

INKEY$ Function

INKEY$

Returns a keyboard character.

Returns a one-character string from the keyboard without having to press <ENTER>. If no key is pressed, a null string (length zero) is returned. Characters typed to INKEY$ are not echoed back to the display.

INKEY$ is invariably put inside some sort of loop. Otherwise, program execution would pass through the line containing INKEY$ before a key could be pressed.

Example

```
A$ = INKEY$
```

Sample Program

```
1Ø A$ = INKEY$
2Ø IF A$ = "" THEN 1Ø
```

this causes the program to wait for a key to be pressed.

INPUT Statement

INPUT [;] ["prompt string";] variable1, variable2,...

Inputs data from the keyboard into one or more variables.

When MBASIC encounters this statement, it stops execution and displays a question mark. This means that the program is waiting for you to type data.

INPUT may specify a list of string or numeric variables, indicating string or numeric values to be input. For instance, INPUT X$, X1, Z$, Z1 calls for you to input a string literal, a number, another string literal, and another number, in that order.

The number of data items you supply must be the same as the number of variables specified. You must separate data items by commas.

Responding to INPUT with too many or too few items, or with the wrong type of value, causes MBASIC to print the message "?Redo from start". No values are assigned until you provide an acceptable response.

If a prompt string is included, MBASIC prints it before the question mark. This helps the person inputting the data to enter it correctly. The prompt string must immediately follow INPUT. It must be enclosed in quotes and followed by a semicolon.

If INPUT is immediately followed by a semicolon, any carriage returns pressed as part of the response are not echoed.

You can enter any valid constant. 2, 1Ø5, 1, 3#, etc. are all valid constants.

Examples

```
INPUT Y%
```

when MBASIC reaches this line, you must type a number and press <ENTER> before the program will continue.

```
INPUT SENTENCE$
```

when MBASIC reaches this line, you must type in a string. The string wouldn't have to be enclosed in quotation marks unless it contained a comma, a colon, or a leading blank.

```
INPUT "ENTER YOUR NAME AND AGE (NAME, AGE)"; N$, A
```

would print a message on the screen which would help the person at the keyboard to enter the right sort of data.

Sample Program

```
5Ø INPUT "HOW MUCH DO YOU WEIGH"; X
6Ø PRINT "ON MARS YOU WOULD WEIGH ABOUT" CINT(X * .38) "POUNDS"
```

INPUT# Statement

INPUT# buffer, variable,...

Inputs data from a sequential disk file and stores it in a program variable.

Buffer is the number used when the file was OPENed for input.

Variable contains the variable name(s) that will be assigned to the item(s) in the file.

With INPUT#, data is input sequentially. That is, when the file is opened, a pointer is set to the beginning of the file. The pointer advances each time data is input. To start reading from the beginning of the file again, you must close the file buffer 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 ten different PRINT# statements. What matters to INPUT# is the position of the terminating characters and the EOF marker.

When inputting data into a variable, MBASIC ignores leading blanks. When the first non-blank character is encountered, MBASIC 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 terminating characters vary, depending on whether MBASIC is inputting to a numeric or string variable.

Numeric values: MBASIC begins input at the first character which is neither a space or a carriage return. It ends input when it encounters a space, carriage return, or a comma.

String values: MBASIC begins input with the first character which is neither a space nor carriage return. It ends input when it encounters a carriage return or comma. One exception to this rule: If the first character is a quotation mark, the string will consist of all characters between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character.

If the end-of-file is reached when a numeric or string item is being INPUT, the INPUT is terminated

Examples

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

sequentially inputs two numeric data items from disk and places them in A and B. Buffer #1 is used.

```
INPUT#4, A$, B$, C$
```

sequentially inputs three string data items from disk and places them in A$, B$, and C$. Buffer #4 is used.

INPUT$ Function

INPUT$(number1 [,number2])

Inputs a string of characters from either the keyboard or a sequential file.

Number1 is the number of characters to be input. It must be a value in the range 1 to 32767. Number2 is a buffer which accesses a sequential input file.

INPUT$(number1) inputs a string of characters from the keyboard. When the program reaches this line, it stops until you (or any operand) type number1 number of characters. (You don't need to press <ENTER> to signify end-of-line). The character(s) you type are not displayed on the screen. Any character, except <BREAK>, is accepted for input. No characters are echoed.

INPUT$(number1,number2) inputs the string from the disk file associated with buffer number.

Examples

```
A$ = INPUT$(5)
```

assigns a string of five keyboard characters to A$. Program execution is halted until the operator types five characters.

```
A$ = INPUT$(11,3)
```

assigns a string of 11 characters to A$. The characters are read from the disk file associated with buffer 3.

Sample Programs

This program shows how you could use INPUT$ to have an operator input their telephone number. By using INPUT$, the operator can type in the number without anyone seeing it on the video display.

```
11Ø LINE INPUT "TYPE IN YOUR AREA CODE "; F$
12Ø PRINT "TYPE IN YOUR PHONE NUMBER--###-####"
13Ø P$ = INPUT$(8)
14Ø F$ = "(" + F$ + ")" + P$
15Ø PRINT F$
```

KEYWORDS

In the program below, line 1ØØ opens a sequential input file (which we assume has been previously created). Line 2ØØ retrieves a string of 7Ø characters from the file and stores them in T$. Line 3ØØ closes the file.

```
1ØØ OPEN "I", 2, "TEST.DAT"
2ØØ T$ = INPUT$(7Ø,2)
3ØØ CLOSE
```

INSTR Function

INSTR([integer,] string1, string2)

Searches for the first occurrence of string2 in string1, and returns the position at which the match is found.

Integer specifies a position in string1. It must be a value in the range 1 to 32767.

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, it returns zero. Note that the entire substring must be contained in the search string, or zero is returned.

Optional integer sets the position for starting the search. If omitted, INSTR starts searching at the first character in string1.

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

The program below uses INSTR to search through the addresses contained in the program's DATA lines. It counts the number of addresses with a specified county zip code (761--) and returns that

number. The zip code is preceded by an asterisk to distinguish it from the other numeric data found in the address.

```
360 RESTORE
370 COUNTER = 0
390 READ ADDRESS$
395 IF ADDRESS = "$END" THEN 410
400 IF INSTR(ADDRESS$, "*761") <> 0 THEN COUNTER = COUNTER +1
405 GOTO 390
410 PRINT "NUMBER OF TARRANT COUNTY, TX ADDRESSES IS" COUNTER:
    END
420 DATA "5950 GORHAM DRIVE, BURLESON, TX *76148"
430 DATA "71 FIRSTFIELD ROAD, GAITHERSBURG, MD *20760"
440 DATA "1000 TWO TANDY CENTER, FORT WORTH, TX *76102"
450 DATA "16633 SOUTH CENTRAL EXPRESSWAY, RICHARDSON, TX
    *75080"
460 DATA "$END"
```

INT Function

INT(number)

Converts number to integer value.

This function returns the largest integer which is not greater than the number.

The result has the same precision as the argument except for the fractional portion. Number is not limited to the range - 32768 to 32767.

Examples

```
PRINT INT(79.89)
```

prints 79.

```
PRINT INT (-12.11)
```

prints -13.

KILL Statement

KILL filename

"Kills" (deletes) filename from disk.

You may KILL any type of disk file. However, if the file is currently OPEN, a "File already open" error occurs. You must CLOSE the file before deleting it.

You can also add a pathname to this statement. If you do not include a pathname, MBASIC searches for the file in the current directory.

Example

```
KILL "FILE.BAS"
```

deletes this file named FILE.BAS from the current directory.

```
KILL "ACCOUNTING/DATA"
```

deletes the file DATA from the directory ACCOUNTING, located in your current directory (we are not showing the entire pathname to simplify this example).

LEFT$ Function

LEFT$(string, integer)

Returns the leftmost integer characters of string.

If integer is equal to or greater than LEN (string), the entire string is returned.

Examples:

```
PRINT LEFT$("BATTLESHIPS", 6)
```

prints BATTLE.

```
PRINT LEFT$("BIG FIERCE DOG", 2Ø)
```

prints BIG FIERCE DOG, since the string is less than 2Ø characters long.

Sample Program

```
74Ø A$ = "TIMOTHY"
75Ø B$ = LEFT$(A$, 3)
76Ø PRINT B$; "--THAT'S SHORT FOR "; A$
RUN
TIM--THAT'S SHORT FOR TIMOTHY
Ok
```

line 75Ø stores the three leftmost characters of A$ and stores them in B$. Line 76Ø prints these three characters, a string, and the original contents of A$.

LEN Function

LEN(string)

Returns the number of characters in string.

Examples

```
X = LEN(SENTENCE$)
```

gets the length of SENTENCE$ and stores it in X.

```
PRINT LEN("CAMBRIDGE") + LEN("BERKELEY")
```

prints 17.

LET — Statement

LET variable = expression

Assigns the value of expression to variable.

MBASIC doesn't require assignment statements to begin with LET, but you might want to use it to be compatible with versions of MBASIC that do require it.

Examples

```
LET A$ = "A ROSE IS A ROSE"
LET B1 = 1.23
LET X  = X - Z1
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant or expression on the right side.

Sample Program

```
55Ø P = 1ØØ1: PRINT "P =" P
56Ø LET P = 2ØØ1: PRINT "NOW P = "P
```

LINE INPUT — Statement

LINE INPUT[;]["prompt message"**;] string variable**

Inputs an entire line (up to 254 characters) from the keyboard.

LINE INPUT is a convenient way to input string data without having to worry about accidental entry of delimiters (commas, quotation marks, carriage returns, etc.).

LINE INPUT is similar to INPUT, except:

- The computer does not display a question mark when waiting for input.
- Each LINE INPUT statement can assign a value to only one variable.
- Commas and quotes can be used 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>. However, if LINE INPUT is immediately followed by a semicolon, pressing <ENTER> does not echo a carriage return at the display.

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$
```

inputs A$ without displaying any prompt.

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

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

You may abort a LINE INPUT statement by pressing <CTRL><C>. MBASIC returns to command level. Typing CONT resumes execution at LINE INPUT.

LINE INPUT is restricted to 254 characters.

LINE INPUT# Statement

LINE INPUT# buffer, variable

Inputs an entire line from a sequential data file to a string variable.

Buffer is the number under which the file was OPENed.

This statement is useful when you want to read an ASCII-format MBASIC program file as data, or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT# reads everything from the first character up to:

- a line feed character
- the end-of-file
- the 32,767th character

Other characters encountered--quotes, commas, leading blanks, line feed, carriage return sequences--are included in the string.

Example

If the data on disk was saved in ASCII format it would look like this:

```
10 CLEAR 500
20 OPEN "I", 1, "PROG"
```

then the statement

```
LINE INPUT#1, A$
```

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

LIST Statement

LIST [startline]-[endline] [,filename]

Lists a program in memory to the display.

Startline specifies the first line to be listed. If omitted, MBASIC starts with the first line in your program.

Endline specifies the last line to be listed. If omitted, MBASIC ends with the last line in your program.

Filename identifies the destination file of the listing. If you omit filename, MBASIC lists the specified lines to the display.

Examples

```
LIST
```

displays the entire program. To stop the automatic scrolling, press <CTRL><S>. This freezes the display. Press <CTRL><Q> to continue.

```
LIST 5Ø
```

displays line 5Ø.

```
LIST 5Ø-85
```

displays lines in the range 5Ø-85.

```
LIST .-
```

displays the program line that has just been entered or edited, and all higher-numbered lines.

```
LIST -227
```

displays all lines up to and including 227.

```
LIST , "EMP"
```

lists an entire program to a file named EMP.

```
LIST 227-
```

Lists all lines starting with line 227 to the end of the file.

LLIST Statement

LLIST [startline]-[endline]

Lists program lines in memory to the printer.

The only difference between LLIST and LIST is that LLIST lists the lines on printer. See LIST.

Examples

LLIST

lists the entire program to the printer. To stop this process, press <CTRL><S>. This causes control to return to MBASIC and therefore aborts LLIST.

LLIST 68-9Ø

prints lines in the range 68-9Ø.

LOAD Statement

LOAD filename, [R]

Loads filename, a MBASIC program, into memory.

The R option causes the program to then run. (LOAD with the R option is equivalent to the command RUN filename, R).

LOAD without the R option wipes out any resident MBASIC program, clears all variables, and CLOSES all OPEN files. LOAD with the R option leaves all OPEN files open and runs the program automatically.

You can use either of these commands inside programs to allow program chaining (one program calling another).

If you attempt to LOAD a non-MBASIC file, a "Direct statement in file" error will occur.

Example

```
LOAD  "PROG1.BAS"
```

LOC Function

LOC(buffer)

Returns the current position in the file.

Buffer is the buffer under which the file was OPENed.

With direct-access files, LOC returns the number of the last record read or written to the direct file.

With sequential-access files, LOC returns the number of 128-byte blocks read from or written to the file since it was OPENed.

For files OPENed to "KYBD:", LOC returns 1 if any characters are ready to be read from standard input. Otherwise, it returns Ø. KYBD: is a "device". For more information on devices, see OPEN and Appendix G.

Example

```
IF LOC(1)>55 THEN END
```

if the current record number is greater than 55, this statement ends program execution.

Sample Program

```
131Ø A$ = "WILLIAM WILSON"
132Ø GET 1
133Ø IF N$ = A$ THEN PRINT "FOUND IN RECORD" LOC(1): CLOSE: END
134Ø GOTO 132Ø
```

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.

LOCK Statement

LOCK [#]buffer [,READ] [,WAIT] [, range]

Restricts access by other programs to the file assigned to buffer in the specified range of records or for the entire file.

Buffer is the buffer number used to OPEN the file.

The READ option, when specified, allows another program to read the locked area of the file; but not write to it. If the READ option is omitted, total access is denied to another program.

The WAIT option tells MBASIC to wait if the file being accessed is already locked. When the file is UNLOCKed, the program continues and the new lock is applied. For this option to work properly, both programs must contain a lock statement. You can interrupt the WAIT by pressing <CTRL><C>. To resume the WAIT, type CONT.

If you do not use the WAIT option, control is returned to the program immediately with an accompanying error message. All of the usual MBASIC error handling can be used to trap and examine this error (i.e., ON ERROR GOTO). If error trapping is not active the **"Permission denied"** error message is returned.

Range specifies the region of the file to be locked and is used for random access files only. The syntax for range is record1 TO record2, where record1 and record2 are inclusive. Record1 is optional, if omitted 1 is used.

Note: If a program is run that does not contain a LOCK statement, and attempts to access a locked file, the program "waits" until the file is unlocked.

Deadlocks

It is possible to get into a deadlock situation when WAITing for a lock request. For example:

```
      Program 1                          Program 2
1Ø OPEN "R", 1, "A", 32          1Ø OPEN "R", 1, "A", 32
2Ø OPEN "R", 2, "B", 32          2Ø OPEN "R", 2, "B", 32
3Ø LOCK 1, 1 TO 5                3Ø LOCK 2, 1 TO 1Ø
4Ø LOCK 2, WAIT, 1 TO 1Ø         4Ø LOCK 1, WAIT, 1 TO 5
```

Program 1 opens Files A and B. File A is locked and File B is locked with the WAIT option. At the same time, Program 2 opens and locks Files A and B, but in this program File A is locked with the WAIT option. If you run these two programs simultaneously, on seperate terminals, the overlapping of LOCK statements with the WAIT option causes a deadlock situation.

TRS-XENIX attempts to detect any deadlock situations. When a deadlock situation is detected error message "Deadlock" is returned.

Multiple LOCKS

Multiple LOCK statements have a cumulative affect. Locking records 1 through 3 and then locking records 1Ø through 1ØØ leaves records 1 through 3 and 1Ø through 1ØØ locked. Locking a record which is already locked has no effect, the record remains locked. A record locked multiple times with different locking arguments (i.e. READ) has the lock characteristics of the last executed LOCK statement. For example if you lock record 1 without the READ option and then relock it with the READ option, record 1 will be locked with the READ option.

It is recommended that you do not open a single file on multiple buffers simultaneously. If it becomes necessary to open a single file multiple times, be aware that locks applied using different buffer numbers against the same file act just as multiple LOCK statements do against a single file. The following program displays this:

```
1Ø OPEN "R", 1, "payroll.dat", 32
2Ø OPEN "R", 2, "payroll.dat", 32
3Ø LOCK #1, 1 TO 3
4Ø LOCK #2, READ, 2 TO 5
```

This program locks records 1 through 5 of the file payroll.dat. In addition, records 2 through 5 are locked in the READ mode. Therefore, even though different buffer numbers were given, both of the specified record locks were executed on the same file. Also, the first buffer that is closed releases all locks against the file, including locks made using another buffer number. If buffer number 2 is closed in the above example, then all locks are released, including records 1 through 3 that were locked for buffer number 1.

Important: Records are locked based upon their position and size in the file. The unit of measure used is the byte. If you open a file multiple times with different record lenghts and apply locks, a portion of a record can become locked.

Sample Program

```
10 OPEN "R", 1, "employee.data", 32
20 LOCK 1, 1 TO 200
30 FOR N% = 1 TO 200
40 GET 1, N%
50 PRINT NAME$
60 PRINT ADDRESS$
70 PRINT TEL$
80 NEXT N%
90 CLOSE 1
```

This program opens the file **employee.data** and locks the records 1 through 200. The information of each record is written out and then the total file is unlocked with the CLOSE statement. If LOCK fails, the **"Permission denied"** error message is returned.

LOF Function

LOF(buffer)

Returns the number of bytes in the file.

Buffer is the number under which the file was OPENed.

LOF is valid for any file. Files OPENed to "LPT1:", "KYBD:", or "SCRN:" always return Ø.

Example

```
Y = LOF(5)
```

assigns to Y the number of bytes in the file accessed by buffer 5.

LOG Function

LOG(number)

Computes the natural logarithm of number.

This is the inverse of the EXP function. The result is always in double precision.

Examples

```
PRINT LOG(3.14159)
```

prints the value 1.1447290411851.

```
Z = 10 * LOG(Ps/P1)
```

performs the indicated calculation and assigns the value to Z.

Sample Program

This program demonstrates the use of LOG. It utilizes a formula taken from space communications research.

```
540 INPUT "DISTANCE SIGNAL MUST TRAVEL (MILES):"; D
550 INPUT "SIGNAL FREQUENCY (GIGAHERTZ)"; F
560 L = 96.58 + (20 * LOG(F)) + (20 * LOG(D))
570 PRINT "SIGNAL STRENGTH LOSS IN FREE SPACE IS" L "DECIBELS."
```

LPOS Function

LPOS(number)

Returns the column position for the device "LPT1:".

Number is a dummy argument.

Example

```
100 IF LPOS(X)>60 THEN LPRINT "Column 60"
```

LPRINT, LPRINT USING — Statement

LPRINT data,...
LPRINT USING format; data,...

Prints data to the printer.

To use these commands, you must execute an END statement in your program. This is because with a multi-user system, MBASIC stores in memory the data you specify. It will not print data until: 1) It executes an END statement or 2) You leave MBASIC by typing SYSTEM.

A better way to print data at the printer is to OPEN the device as a file by specifying "LPT1:" as the filename. Print the data to the device, then close "LPT1:". This spools the data file to the printer.

Examples

```
LPRINT (A * 2)/3
```

prints the value of expression (A * 2)/3.

```
LPRINT TAB(50) "TABBED 50"
```

moves the line printer carriage to TAB position 50 and prints "TABBED 50". (Refer to the TAB function).

```
LPRINT USING "#####.#"; 2.17
```

sends the formatted value ØØØØ2.2 to the line printer.

See PRINT and PRINT USING for more information.

Sample Program

This illustrates how to use a file for printing data to the printer.

```
5  OPEN "O", 1, "LPT1:"
10 PRINT #1, "Name"
20 PRINT #1, "Address"
30 CLOSE #1
```

See OPEN for more information.

LSET Statement

LSET field name = data

Sets data in a direct-access buffer field name.

Before using LSET, you must have used FIELD to set up buffer fields.

See also the chapter on "Disk Files", OPEN, CLOSE, FIELD, GET, PUT, and RSET.

Example

Suppose NM$ and AD$ have been defined as field names for a direct access file buffer. NM$ has a length of 18 characters; AD$ has a length of 25 characters. The statements

```
LSET NM$ = "JIM CRICKET, JR."
LSET AD$ = "2ØØØ EAST PECAN ST."
```

set the data in the buffer as follows:

JIMØCRICKET,JR.ØØØ 2ØØØØEASTØPECANØST.ØØØØØØ

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

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

MEM Function

MEM

Returns the amount of memory.

MEM performs the same function as FRE. MEM returns the number of unused bytes in memory.

This function may be used in the immediate mode to see how much space a resident program occupies, or it may be used inside a program to avert "Out of memory" errors. MEM requires no argument. The internal program stack is not part of this total.

Example

```
PRINT MEM
```

Enter this command (in the immediate mode, no line number is needed). The number returned indicates the amount of leftover memory, that is, memory not being used to store programs, variables, strings, or the stack.

Sample Program

```
161Ø IF MEM < 8Ø THEN 163Ø
162Ø DIM A(15)
163Ø REM       PROGRAM CONTINUES HERE
```

If fewer than 8Ø bytes of memory are left, control switches to another part of the program. Otherwise, an array of 16 elements is created.

MERGE Statement

MERGE filename

Loads filename, a MBASIC program, and merges it with the program currently in memory.

Filename specifies a MBASIC file in ASCII format (a program saved with the A option). If filename is a constant, it must be enclosed in quotes.

Program lines in the disk program are inserted into the resident program in sequential order. For example, suppose that three of the lines from the disk program are numbered 75,85 and 90, and three of the lines from the current program are numbered 70, 80, and 90. When MERGE is used on the two programs, this portion of the new program will be numbered 70, 75, 80, 85, 90,.

If line numbers on the disk program coincide with line numbers in the resident program, the disk program's lines replace the resident program's lines.

MERGE closes all files and clears all variables. Upon completion, MBASIC returns to the command mode.

Example

Suppose you have a MBASIC program on disk, PROG2.TXT (saved in ASCII), which you want to merge with the program you've been working on in memory. Then we use:

```
MERGE  "PROG2.TXT"
```

merges the two programs.

Sample Programs

MERGE provides a convenient means of putting program modules together. For example, an often-used set of MBASIC subroutines can be tacked onto a variety of programs with this command.

Suppose the following program is in memory:

```
80  REM   MAIN PROGRAM
90  GOSUB 1000
100 REM   PROGRAM LINE
```

```
11Ø REM   PROGRAM LINE
12Ø REM   PROGRAM LINE
13Ø END
```

And suppose the following subroutine, SUB.TXT, is stored on disk in ASCII format:

```
1ØØØ REM  BEGINNING OF SUBROUTINE
1Ø1Ø REM  SUBROUTINE LINE
1Ø2Ø REM  SUBROUTINE LINE
1Ø3Ø REM  SUBROUTINE LINE
1Ø4Ø RETURN
```

You can MERGE the subroutine with the main program with:

```
MERGE "SUB.TXT"
```

and the new program in memory is:

```
8Ø   REM  MAIN PROGRAM
9Ø   GOSUB  1ØØØ
1ØØ  REM  PROGRAM LINE
11Ø  REM  PROGRAM LINE
12Ø  REM  PROGRAM LINE
13Ø  END
1ØØØ REM  BEGINNING OF SUBROUTINE
1Ø1Ø REM  SUBROUTINE LINE
1Ø2Ø REM  SUBROUTINE LINE
1Ø3Ø REM  SUBROUTINE LINE
1Ø4Ø RETURN
```

MID$ Statement

MID$(oldstring, position, [length]) = replacement string

Replaces a portion of a oldstring with replacement string.

Oldstring is the variable name of the string you want to change.

Position is a number specifying the position of the first character to be changed.

Length is a number specifying the number of characters to be replaced. If omitted, all of replacement string is used.

Replacement string is the string to replace a portion of oldstring. The length of the resultant string is always the same as the original string. If replacement string is shorter than length, the entire replacement string is used.

Examples:

```
A$ = "LINCOLN"

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

returns LI1234N.

```
MID$ (A$, 5) = "Ø1": PRINT A$
```

returns LINCØ1N.

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

returns ***COLN.

MID$ Function

MID$(string, integer [,number])

Returns a substring of string, beginning with the integer character.

If integer is greater than the number of characters in string, MID$ returns a null string.

Number is the number of characters in the substring. If omitted, MBASIC returns all right most characters, beginning with the character at position integer..

Examples

If A$ = "WEATHERFORD" then

```
PRINT MID$(A$, 3, 2)
```

prints AT.

```
F$ = MID$(A$, 3)
```

puts ATHERFORD into F$.

Sample Program

```
200 INPUT "AREA CODE AND NUMBER (NNN-NNN-NNNN)"; PH$
210 EX$ = MID$(PH$, 5, 3)
220 PRINT "NUMBER IS IN THE " EX$ " EXCHANGE."
```

The first three digits of a local phone number are sometimes called the exchange of the number. This program looks at a complete phone number (area code, exchange, last four digits) and picks out the exchange of that number.

MKD$, MKI$, MKS$ Function

MKI$(integer expression)
MKS$(single-precision expression)
MKD$(double-precision expression)

Convert numeric values to string values.

Any numeric value that is placed in a direct file buffer with an LSET or RSET statement must be converted to a string.

These three functions are the inverse of CVD, CVI, and CVS. 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.

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

Example

```
LSET AVG$ = MKS$(Ø.123)
```

Sample Program

```
135Ø OPEN "D", 1, "TEST.DAT"
136Ø FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
137Ø LSET I1$ = MKI$(3ØØØ)
138Ø LSET I2$ = MKS$(3ØØØ.1)
139Ø LSET I3$ = MKD$(3ØØØ.ØØØØ1)
14ØØ PUT 1
141Ø CLOSE
```

For a program that retrieves the data from TEST.DAT, see CVD/CVI/CVS.

NAME Statement

NAME old filename **AS** new filename

Renames old filename as new filename.

With this statement, the data in the file is left unchanged.

Example

```
NAME "FILE" AS "FILE.OLD"
```

renames FILE as FILE.OLD.

```
NAME B$ AS A$
```

Renames B$ as A$.

NEW Statement

NEW

Deletes the program currently in memory and clears all variables.

NEW returns you to the command mode.

Example

```
NEW
```

OCT$ Function

OCT$(number)

Computes the octal value of number.

OCT$ returns a string which represents the octal value of number. The value returned is like any other string--it cannot be used in a numeric expression.

Examples

```
PRINT OCT$(3Ø),  OCT$(5Ø),  OCT$(9Ø)
```

prints the following strings:

```
36        62        132
```

```
Y$ = OCT$(X/84)
```

Y$ is a string representation of the integer quotient X/84 to base 8.

ON ERROR GOTO Statement

ON ERROR GOTO line

Transfers control to line if an error occurs.

This lets your program "recover" from an error and continue execution. (Normally, you have a particular type of error in mind when you use the ON ERROR GOTO statement).

ON ERROR GOTO has no effect unless it is executed before the error occurs. To disable it, execute an ON ERROR GOTO Ø. If you use ON ERROR GOTO Ø inside an error-trapping routine, MBASIC stops execution and prints an error message.

The error-handling routine must be terminated by a RESUME statement. See RESUME.

Example

```
1Ø ON ERROR GOTO 15ØØ
```

branches program control to line 15ØØ if an error occurs anywhere after line 1Ø.

ON...GOSUB Statement

ON number **GOSUB** line1, line2,...

Goes to a subroutine at the line specified by the value of number.

Number must be between Ø and 255. For example, if number's value is three, the third line number on the list is the destination of the branch.

If number's value is zero or greater than the number of items on the list (but less than or equal to 255), MBASIC continues with the next executable statement. If number is negative or greater than 255, an "Illegal function call" error occurs.

Example

```
ON Y GOSUB 1ØØØ, 2ØØØ, 3ØØØ
```

if Y = 1, the subroutine beginning at 1ØØØ is called. If Y = 2, the subroutine at 2ØØØ is called. If Y = 3, the subroutine at 3ØØØ is called.

Sample Program

```
43Ø INPUT "CHOOSE 1, 2, OR 3" ; I
44Ø ON I GOSUB 5ØØ, 6ØØ, 7ØØ
45Ø END
5ØØ PRINT "SUBROUTINE #1": RETURN
6ØØ PRINT "SUBROUTINE #2": RETURN
7ØØ PRINT "SUBROUTINE #3": RETURN
```

ON...GOTO Statement

ON number **GOTO** line, line,...

Goes to the line specified by the value of number.

Number is a numeric expression between Ø and 255.

This statement is very similar to ON...GOSUB. However, instead of branching to a subroutine, it branches control to another program line.

The value of number determines to which line the program will branch. For example, if the value is four, the fourth line number on the list is the destination of the branch. If there is no fourth line number, control passes to the next statement in the program.

If the value of expression is negative or greater than 255, an "Illegal function call" error occurs. Any amount of line numbers may be included after GOTO.

Example

```
ON MI GOTO 15Ø, 16Ø, 17Ø, 15Ø, 18Ø
```

tells MBASIC to "Evaluate MI,
if the value of MI equals one then go to line 15Ø;
if it equals two, then go to 16Ø;
if it equals three, then go to 17Ø;
if it equals four, then go to 15Ø;
if it equals five, then go to 18Ø;

if the value of MI doesn't equal any of the numbers one through five, advance to the next statement in the program".

OPEN Statement

OPEN mode, buffer, filename [,record length]

Mode is a string expression whose first character is one of the following:

O (opens the file for output)
I (opens the file for input)
A (opens a sequential file for output, with the pointer positioned at the end-of-data)
R (opens a random (direct)-access file)
D (opens a random (direct)-access file)

or

OPEN filename [FOR mode] AS buffer [LEN=record length]

Mode is a string expression whose value is:

INPUT (opens the file for input)
OUTPUT (opens the file for output)
APPEND (opens a sequential file for output, with the pointer positioned at the end-of-data)
RANDOM (opens a random (direct)-access file. If mode is omitted RANDOM is used.)

Opens a disk file.

Buffer is an integer between 1 and 255. It specifies the area in memory that will be used to access the file. This number will be associated with the file for as long as the file is OPEN.

Filename specifies a TRS-XENIX file.

Record length is an integer which sets the record length for direct-access files. This length can be as large as 32767 bytes. The default is 256 bytes.

Examples

```
OPEN "O", 1, "CLIENTS.TXT"
```

opens the file CLIENTS.TXT for output. Buffer 1 is used. If the file does not exist, it is created. If it already exists, its previous contents are lost.

```
OPEN "D", 2, "DATA.BAS"
```

opens the file DATA.BAS in random-access mode. Buffer 2 is used.

```
OPEN "TEST.BAS" FOR OUTPUT AS 5
```

opens the file TEXT.BAS for output. Buffer 5 is used.

See the chapter on "Disk Files" for programming information.

OPTION BASE Statement

OPTION BASE n

Sets n as the minimum value for an array subscript.

N may be 1 or Ø. The default is Ø.

If the statement

OPTION BASE 1

is executed, the lowest value an array subscript may have is one.

PEEK Function

PEEK(memory location)

Returns a byte from memory location.

The memory location of MBASIC's User-readable data includes MBASIC's variables and stack, User's program, User's variables, strings, and File Data Blocks.

The value returned is an integer between Ø and 255.

PEEK is the complementary function of the statement POKE.

Example

```
A = PEEK (&H5AØØ)
```

POKE Statement

POKE memory location, data byte

Writes data byte into memory location.

Data byte must be a decimal number between Ø and 255.
Memory location must be inside MBASIC's User-writable data space which includes User's variables, strings, and File Data Blocks.

POKE is the complementary statement of PEEK. The argument to PEEK is a memory location from which a byte is to be read.

PEEK and POKE can be used for storing data efficiently.

It is not recommended to use PEEK and POKE because you run the risk of destroying your program and data and it also prevents software portability.

Example

```
1Ø POKE &H5AØØ, &HFF
```

12-27-84 3:30 PM
JM called Texas to inquire about poking characters on to the screen. Told that the model 16B does not handle poke. Tried example in book and nothing visible happened.

POS Function

POS(dummy number)

Returns the column position for the device "SCRN:"

Number is a dummy argument.

POS returns a number from 1 to 8Ø indicating the current column position of the cursor on the display.

Example

```
PRINT TAB(4Ø) POS(Ø)
```

Sample Program

```
15Ø CLS
16Ø A$ = INKEY$
17Ø IF A$ = "" THEN 16Ø
18Ø IF POS(X) > 7Ø THEN IF A$ = CHR$(32) THEN A$ = CHR$(13)
19Ø PRINT A$;
2ØØ GOTO 16Ø
```

See Appendix G for more information.

PRINT Statement

PRINT data,...

Prints numeric or string data on the display.

MBASIC prints the values of the data items you list in this statement.

You may separate the data items by commas or semicolons. If you use commas, the cursor automatically advances to the next tab position before printing the next item. (MBASIC divides each line into five tab positions, at columns Ø, 14, 28, 42, and 56). If you use semicolons, it prints the items without any spaces between them. For example,

```
PRINT C$;T$,M$
```

works the same way as

```
PRINT C$T$M$
```

A semicolon or comma at the end of a line causes the next PRINT statement to begin printing where the last one left off. If you do no trailing punctuation is used with PRINT, the cursor drops down to the beginning of the next line.

Single-precision numbers with six or fewer digits that can be accurately represented in ordinary (rather than exponential) format, are printed in ordinary format.

Double-precision numbers with 14 or fewer digits that can be accurately represented in ordinary format, are printed using the ordinary format.

MBASIC prints positive numbers with a leading blank. It prints all numbers with a trailing blank.

To insert strings into this statement, surround them with quotation marks.

Example

```
PRINT "DO"; "NOT"; "LEAVE"; "SPACES"; "BETWEEN";
"THESE"; "WORDS"
```

prints on the display: DONOTLEAVESPACESBETWEENTHESEWORDS

Sample Program

```
60 INPUT "ENTER THIS YEAR"; Y
70 INPUT "ENTER YOUR AGE";A
80 INPUT "ENTER A YEAR IN THE FUTURE";F
90 N=A+(F-Y)
100 PRINT "IN THE YEAR"F"YOU WILL BE"N"YEARS OLD"
RUN
```

Since F and N are positive numbers, PRINT inserts a space before and after them; therefore, your display should look similar to this (depending on your input):

```
IN THE YEAR 2004 YOU WILL BE 46 YEARS OLD
```

If we had separated each expression in line 100 by a comma,

```
100 PRINT "IN THE YEAR",F,"YOU WILL BE",N,"YEARS OLD"
```

our display would show:

```
IN THE YEAR    2004      YOU WILL BE          46      YEARS OLD
```

MBASIC moved to the next tab position after printing each data item.

PRINT USING Statement

PRINT USING format; data item,...

Prints data items using a format specified by you.

Format consists of one or more field specifier(s), or any alphanumeric character.

Data item may be string and/or numeric value(s).

This statement is especially useful for printing report headings, accounting reports, checks, or any other documents which require a specific format.

With PRINT USING, you may use certain characters (field specifiers) to format the field. These field specifiers are described below. They are followed by sample program lines and their output to the screen.

Specifiers for String Fields:

! Print the first character in the string only.

```
PRINT USING "!"; "PERSONNEL"
P
```

\spaces\ Print 2+ n characters from the string. If you type the backslashes without any spaces, MBASIC prints two characters; with one space, MBASIC prints three characters, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified and padded with spaces on the right.

```
PRINT USING "\   \"; "PERSONNEL"
(three spaces between the backslashes)
PERSO
```

& Print the string without modifications.

```
10 A$="TAKE":B$="RACE"
20 PRINT USING "!";A$
30 PRINT USING "&";B$

TRACE
```

Specifiers for Numeric Fields:

Print the same amount of digit positions as number signs (#). If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces). Numbers are rounded as necessary. You may insert a decimal point at any position. In that case, the digits preceding the decimal point are always printed (as zero, if necessary).

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

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

If the number of digits specified exceeds 24, an "Illegal function call" occurs.

```
PRINT USING "##.##";.75
Ø.75

PRINT USING "###.##";876.567
876.57
```

+ Print the sign of the number. The plus sign may be typed at the beginning or at the end of the format string.

```
PRINT USING "+##.##    "; -98.45,3.5Ø,22.22,-.9   -98.45
+3.5Ø   +22.22   -Ø.9Ø

PRINT USING "##.##+    "; -98.45,3.5Ø,22.22,-.9  98.54-
3.5Ø+   22.22+   Ø.9Ø-
```

(Note the use of spaces at the end of a format string to separate printed values).

- Print a negative sign after negative numbers (and a space after positive numbers).

```
PRINT USING "###.#-"; -768.66Ø
768.7-
```

** Fill leading spaces with asterisks. The two asterisks also establish two more positions in the field.

```
PRINT USING "**####"; 44.Ø
****44
```

$$ Print a dollar sign immediately before the number. This specifies two more digit positions, one of which is the dollar sign.

```
PRINT USING "$$##.##"; 112.789Ø
 $112.79
```

**$ Fill leading spaces with asterisks and print a dollar sign immediately before the number.

```
PRINT USING "**$##.##"; 8.333
***$8.33
```

, Print a comma before every third digit to the left of the decimal point. The comma establishes another digit position.

```
PRINT USING "####,.##"; 1234.5
1,234.5Ø
```

^^^^ Print in exponential format. The four carats are placed after the digit position characters. You may specify any decimal point position.

```
PRINT USING ".####^^^^"; 888888
.8889E+Ø6
```

_ Print next character as a literal character.

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

Sample Program

```
42Ø CLS: A$ = "**$##,######.## DOLLARS"
43Ø INPUT "WHAT IS YOUR FIRST NAME"; F$
44Ø INPUT "WHAT IS YOUR MIDDLE NAME"; M$
45Ø INPUT "WHAT IS YOUR LAST NAME"; L$
46Ø INPUT "ENTER AMOUNT PAYABLE"; P
47Ø CLS : PRINT "PAY TO THE ORDER OF  ";
```

```
48Ø PRINT USING "!! !!  "; F$; "."; M$; ".";
49Ø PRINT L$
5ØØ PRINT :PRINT USING A$; P
```

In line 48Ø, each ! picks up the first character of one of the following strings (F$, ".", M$, and "." again). Notice the two spaces in "!!Ø!!Ø". These two spaces insert the appropriate spaces after the initials of the name (see below). Also notice the use of the variables A$ for format and P for item list in line 5ØØ. Any serious use of the PRINT USING statement would probably require the use of variables at least for item list rather than constants. (We've used constants in our examples for the sake of better illustration).

When the program above is run, the output should look something like this:

```
WHAT IS YOUR FIRST NAME? JOHN
WHAT IS YOUR MIDDLE NAME? PAUL
WHAT IS YOUR LAST NAME? JONES
ENTER AMOUNT PAYABLE? 12345.6
PAY TO THE ORDER OF J. P. JONES

*****$12,435.6Ø DOLLARS
```

PRINT @ Statement

PRINT@ location,
PRINT@ (row, column),

Specifies exactly where printing is to begin.

The location specified must be a number between Ø and 1919. It can also be a a pair of numbers (r, c), where 24>r=>Ø and 79=>c=>Ø.

Whenever you instruct MBASIC to PRINT @ the bottom line of the display, it generates an automatic line feed; everything on the display moves up one line. To suppress this automatic line feed, use a trailing semicolon at the end of the statement.

Examples

```
PRINT @ (11,39), "*"
```

prints an asterisk in the middle of the display.

```
PRINT @ Ø, "*"
```

prints an asterisk at the top left corner of the display.

PRINT TAB(n) Statement

Moves the cursor to the n position on the current line (or on succeeding lines if you specify TAB positions greater than 79).

TAB may be used more than once in a print list.

Since numerical expressions may be used to specify a TAB position, TAB can be very useful in creating tables, graphs of mathematical functions, etc.

TAB can't be used to move the cursor to the left. If the cursor is to the right of the specified position, the TAB statement will simply be ignored.

Example

```
PRINT TAB(5)  "TABBED 5"; TAB(25) "TABBED 25"
```

Notice that no punctuation is needed after the TAB modifiers.

Sample Program

```
22Ø CLS
23Ø PRINT TAB(2) "CATALOG NO." TAB(16) "DESCRIPTION OF ITEM";
24Ø PRINT TAB (39) "QUANTITY"; TAB(51) "PRICE PER ITEM";
245 PRINT TAB (69) "TOTAL PRICE"
```

PRINT# Statement

PRINT# buffer, item1, item2,...

Prints data items in a sequential disk file.

Buffer is the buffer number used to OPEN the file for input.

When you first OPEN a file for sequential output, MBASIC sets a pointer to the beginning of the file--that's where PRINT# starts printing the values of the items. At the end of each PRINT# operation, the pointer advances, so values are written in sequence.

A PRINT# statement creates a disk image similar to what a PRINT to the display creates on the screen. For this reason, make sure to delimit the data so that it will be input correctly from the disk.

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

Examples

If A = 123.45

```
PRINT#1,A
```

writes this nine-byte character sequence onto disk:

Ø123.45Ø carriage return

The punctuation in the PRINT list is very important. Unquoted commas and semicolons have the same effect as they do in regular PRINT statements to the display. For example, if A = 23ØØ and B = 1.3Ø3, then

```
PRINT#1, A,B
<ENTER>
```

writes the data on disk as

Ø 23ØØ ØØØØØØØØØØ 1.3Ø3Ø carriage return

The comma between A and B in the PRINT# list causes 1Ø extra spaces in the disk file. Generally you wouldn't want to use up disk space in this way, so you should use semicolons instead of commas.

Files can be written in a carefully controlled format using PRINT# USING. You can also use this option to control how many characters of a value are written to disk.

For example, suppose A$ = "LUDWIG", B$ = "VAN", and C$ = "BEETHOVEN". Then the statement

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

would write the data in nickname form:

```
L.V.BEET
```

(In this case, we didn't want to add any explicit delimiters). See PRINT USING for more information on the USING option.

PUT Statement

PUT buffer, [record]

Puts a record in a direct-access disk file.

Buffer is the same buffer used to OPEN the file.

Record is the record number you want to PUT into the file. It is an integer between 1 and 32,767. If omitted, the current record number is used.

This statement moves data from the buffer into a specified place in the file.

The first time you access a file via a particular buffer, the next record is set equal to one. (The next record is the record whose number is one greater than the last record accessed).

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

The first time you use PUT after OPENing a file, you must specify the record.

See the chapter on "Disk Files" for programming information.

```
PUT 1
```

writes the next record from buffer 1 to a direct-access file.

```
PUT 1, 25
```

writes record 25 from buffer 1 to a direct-access file.

RANDOMIZE Function

RANDOMIZE [expression]

Reseeds the random number generator.

If your program uses the RND function, every time you load it, MBASIC generates the same sequence of pseudorandom numbers. Therefore, you may want to put RANDOMIZE at the beginning of the program. This will help ensure that you get a different sequence of pseudorandom numbers each time you run the program.

If expression is omitted, MBASIC prompts for a seed value:

```
Random Number Seed (-32768 to 32767)?
```

before executing RANDOMIZE.

RANDOMIZE needs to execute just once.

Sample Program

```
60Ø  CLS: RANDOMIZE
61Ø  INPUT "PICK A NUMBER BETWEEN 1 AND 5"; A
62Ø  B = RND(5)
63Ø  IF A = B THEN 65Ø
64Ø  PRINT "YOU LOSE, THE ANSWER IS" B "--TRY AGAIN."
645  GOTO 61Ø
65Ø  PRINT "YOU PICKED THE RIGHT NUMBER -- YOU WIN!": GOTO 61Ø
```

READ Statement

READ variable1, variable2,...

Reads values from a DATA statement and assigns them to variables.

MBASIC assigns values from the DATA statement on a one-to-one basis. The first time READ is executed, the first value in the first DATA statement is used; the second time, the second value is used, and so on.

A single READ may access one or more DATA statements (each DATA statement is accessed in order), or several READs may access the same DATA statement.

The values read must agree with the variable types specified in list of variables, otherwise, a "Syntax error" occurs. If the number of variables in list of variables exceeds the number of elements in the DATA statement(s), an "Out of data" error message is printed.

If the number of variables specified is lower than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element.

Example

```
READ T
```

reads a numeric value from a DATA statement and assigns it to variable "T".

Sample Program

This program illustrates a common application for the READ and DATA statements.

```
40  PRINT "NAME", "AGE"
50  READ N$
60  IF N$="END" THEN PRINT "END OF LIST": END
70  READ AGE
80  IF AGE<18 THEN PRINT N$, AGE
90  GOTO 50
100 DATA "SMITH, JOHN", 30, "ANDERSON, T.M.", 20
110 DATA "JONES, BILL", 15, "DOE, SALLY", 21
120 DATA "COLLINS, W.P.", 17: END
```

REM Statement

REM

Inserts a remark line in a program.

REM instructs the computer to ignore the rest of the program line. This allows you to insert remarks into your program for documentation. Then, when you look at a listing of your program, or someone else does, it will be easier to figure it out.

If REM is used in a multi-statement program line, it must be the last statement in the line.

You may use an apostrophe (') as an abbreviation for :REM.

Sample Program

```
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
```

OR

```
120 FOR I=1 TO 20          'CALCULATE AVERAGE VELOCITY
130 SUM=SUM + V(I)
140 NEXT I
```

RENUM Statement

RENUM [new line], [line], [increment]

Renumbers a program, starting at line, using new line as the first new line and increment for the new sequence.

If you omit new line, MBASIC starts numbering at line 10.
If you omit the line, it renumbers the entire program.
If you omit increment, it jumps 10 numbers between lines.

RENUM also changes all line number references appearing after GOTO, GOSUB, THEN, ON ...GOTO, ON...GOSUB, ON ERROR GOTO, RESUME, and ERL[relational operator].

Examples

RENUM

renumbers the entire resident program, incrementing by 10's. The new number of the first line will be 10.

RENUM 600, 5000, 100

renumbers all lines numbered from 5000 up. The first renumbered line will become 600, and an increment of 100 will be used between subsequent lines.

RENUM 10000, 1000

renumbers line 1000 and all higher-numbered lines. The first renumbered line will become line 10000. An increment of 10 will be used between subsequent line numbers.

RENUM 100, , 100

renumbers the entire program, starting with a new line number of 100, and incrementing by 100's. Notice that the commas must be retained even though the middle argument is gone.

Error Conditions

1. RENUM cannot be used to change the order of program lines. For example, if the original program has lines numbered 10,20, and 30, then the command:

   RENUM 15, 30

   is illegal, since the result would be to move the third line of the program ahead of the second. In this case, and FC (illegal function call) error occurs, and the original program is left unchanged.

2. RENUM will not create new line numbers greater than 65529. Instead, an FC error occurs, and the original program is left unchanged.

3. If an undefined line number is used inside your original program, RENUM prints a warning message, "undefined line XXXX in YYYY", where XXXX is the original line number reference and YYYY is the original number of the line containing XXXX. Note that RENUM renumbers the program in spite of this warning message. It does not change the incorrect line number reference, but it does renumber YYYY, according to the parameters in your RENUM command.

RESTORE Statement

RESTORE [line]

Restores a program's access to previously-read DATA statements.

This lets your program re-use the same DATA lines.
If line is specified, the next READ statement accesses the first item in the specified DATA statement.

Sample Program

```
16Ø  READ X$
17Ø  RESTORE
18Ø  READ Y$
19Ø  PRINT X$, Y$
2ØØ  DATA THIS IS THE FIRST ITEM, AND THIS IS THE
     SECOND
```

When this program is run,

```
THIS IS THE FIRST ITEM      THIS IS THE FIRST ITEM
```

is printed on the display. Because of the RESTORE statement in line 17Ø, the second READ statement starts over with the first DATA item.

RESUME Statement

RESUME [line]
RESUME NEXT

Resumes program execution after an error-handling routine.

RESUME without an argument and RESUME Ø both cause the computer to return to the statement in which the error occurred.

RESUME line causes the computer to branch to the specified line number.

RESUME NEXT causes the computer to branch to the statement following the point at which the error occurred.

A RESUME that is not in an error-handling routine causes a "RESUME without error" message.

Examples

```
RESUME
```

if an error has occurred, this line transfers program control to the statement in which it occurred.

```
RESUME 1Ø
```

if an error has occurred, transfers control to line 1Ø.

Sample Program

```
1Ø ON ERROR GOTO 9ØØ
.
.
.
9ØØ IF (ERR=23Ø) AND(ERL=9Ø) THEN PRINT "TRY AGAIN" :
RESUME 8Ø
```

Refer to Appendix A for more on the Error Codes.

RETURN Statement

RETURN

Returns control to the line immediately following the most recently executed GOSUB.

If the program encounters a RETURN statement without execution of a matching GOSUB, an error occurs.

Sample Program

```
33Ø PRINT "THIS PROGRAM FINDS THE AREA OF A CIRCLE"
34Ø INPUT "TYPE IN A VALUE FOR THE RADIUS"; R
35Ø GOSUB 37Ø
36Ø PRINT "AREA IS" ; A: END
37Ø A = 3.14 * R * R
38Ø RETURN
```

RIGHT$ Function

RIGHT$(string, number)

Returns the rightmost number characters of string.

RIGHT$ returns the last number characters of string. If LEN (string) is less than or equal to number, the entire string is returned.

Examples:

```
PRINT RIGHT$("WATERMELON", 5)
```

prints MELON.

```
PRINT RIGHT$("MILKY WAY", 25)
```

prints MILKY WAY.

Sample Program

```
85Ø RESTORE: ON ERROR GOTO 88Ø
86Ø READ COMPANY$
87Ø PRINT  RIGHT$(COMPANY$, 2), : GOTO 86Ø
88Ø END
89Ø DATA "BECHMAN LUMBER COMPANY, SEATTLE, WA"
9ØØ DATA "ED NORTON SEWER SERVICE, BROOKLYN, NY"
91Ø DATA "HAMMON MANUFACTURING COMPANY, HAMMOND, IN"
```

This program prints the name of the state in which each company is located.

RND Function

RND(number)

Generates a pseudorandom number between Ø and number.

Number must be greater than or equal to Ø and less than 32768.

RND produces a pseudorandom number using the current "seed" number. MBASIC generates the seed internally, therefore, it is not accessible to the user. RND may be used to produce random numbers between Ø and 1, or random integers greater than Ø, depending on the argument.

RND(Ø) returns a double-precision value between Ø and 1, RND(number) returns an integer between 1 and number. For example, RND(55) returns a pseudorandom integer between 1 and 55. RND(55.5) returns a pseudorandom number between 1 and 56 (the argument is rounded). If number is negative, a function call error occurs.

Examples

```
A = RND(2)
```

assigns A a value of 1 or 2.

```
A = RND(45)
```

assigns A a random integer between 1 and 45.

```
PRINT RND (Ø)
```

prints a decimal fraction between Ø and 1.

RSET Statement

RSET field name = data

Sets data in a direct-access buffer field.

This statement is similar to LSET. The difference is that with RSET, data is right-justified in the buffer.

See LSET for details.

RUN Statement

RUN [line]
RUN"filename, [R]

Runs a program.

RUN followed by a line or nothing at all simply executes the program in memory, starting at line or at the beginning of the program.

RUN followed by a filename loads a program from disk and then runs it. Any resident MBASIC program is replaced by the new program.

Optional R leaves all previously OPEN files open. If omitted, MBASIC closes all open files.

RUN automatically CLEARS all variables.

Examples

```
RUN
```

starts execution at lowest line number.

```
RUN 10Ø
```

starts execution at line 10Ø.

```
RUN "PROGRAM.A"
```

loads and executes PROGRAM.A.

```
RUN "EDITDATA", R
```

loads and executes EDITDATA, leaving OPEN files open.

SAVE Statement

SAVE filename, [A], [P]

Saves a program in a disk file under filename.

If filename already exists, its contents will be lost as the file is re-created.

SAVE without the A option saves the program in a compressed format. This takes up less disk space. It also helps SAVEs and LOADs to be performed faster. MBASIC programs are stored in RAM using compressed format.

Using the A option causes the program to be saved in ASCII format. This takes up more disk space. However, the ASCII format allows you to MERGE this program later on. Also, data programs which will be read by other programs must usually be in ASCII.

For compressed-format programs, a useful convention is to use the extension .BAS. For ASCII-format programs, use .TXT.

The P option protects the file by saving it in an encoded binary format.

Examples

```
SAVE  "FILE1.BAS.JOHN"
```

saves the resident MBASIC program in compressed format. The file name is FILE1; the extension is .BAS.

```
SAVE "MATHPAK.TXT", A
```

saves the resident program in ASCII form, using the name MATHPAK.TXT, on the first non-write-protected drive.

SGN Function

SGN(number)

Determines number's sign.

If number is a negative number, SGN returns -1.
If number is a positive number, SGN returns 1.
If number is zero, SGN returns Ø.

Examples

```
Y = SGN(A * B)
```

determines what the sign of the expression A*B is, and passes the appropriate number (-1,Ø,1) to Y.

```
ON SGN(X)+2 GOTO 1ØØ, 2ØØ, 3ØØ
```

branches to 1ØØ if X is negative, 2ØØ if X is Ø, and 3ØØ if X is positive.

SHELL Function

SHELL("command")

Returns the process ID of TRS-XENIX command. This process executes command. Control is returned to MBASIC, without waiting for the process to terminate.

Example

```
X = SHELL("date")
```

stores the process ID in x and returns to MBASIC. A few seconds later your screen displays:

```
Sat Jan 29 Ø6:36:Ø6 EST 1983
```

Note: Only numeric variables can store this value.

SHELL Statement

SHELL ["command"]

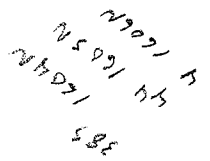
Executes a TRS-XENIX command and returns to MBASIC.

If command is omitted, SHELL halts the execution of MBASIC and executes the TRS-XENIX SHELL program. To exit the SHELL press <CTRL><D> and return to MBASIC.

Example

```
SHELL "ls"
```

displays the listing of your current directory, then returns control to MBASIC.



SIN Function

SIN(number)

Computes the sine of number.

Number must be in radians. To obtain the sine of number when number is in degrees, use SIN(number * .Ø1745329). The result is always double precision.

Examples

```
PRINT SIN(7.96)
```

prints .9943853150281.

Sample Program

```
66Ø INPUT "ANGLE IN DEGREES"; A
67Ø PRINT "SINE IS"; SIN(A * .Ø1745329)
```

SPACE$ Function

SPACE$(number)

Returns a string of number spaces.

Number must be in the range Ø to 32767.

Example

```
PRINT "DESCRIPTION" SPACE$(4) "TYPE" SPACE$(9)
"QUANTITY"
```

prints DESCRIPTION, four spaces, TYPE, nine spaces, QUANTITY.

Sample Program

```
92Ø  PRINT "Here"
93Ø  PRINT SPACE$(13) "is"
94Ø  PRINT SPACE$(26) "an"
95Ø  PRINT SPACE$(39) "example"
96Ø  PRINT SPACE$(52) "of"
97Ø  PRINT SPACE$(65) "SPACE$"
```

SPC Function

SPC(number)

Returns a string of number blanks.

Number is in the range Ø to 32767. SPC does not use string space. The left parenthesis must immediately follow SPC.

SPC may only be used with PRINT, LPRINT, or PRINT# .

Example

```
PRINT "HELLO" SPC(15) "THERE"
```

prints HELLO, 15 spaces, THERE

SQR Function

SQR(number)

Calculates the square root of number.

The number must be greater than zero.

The result is always double precision.

Example

```
PRINT SQR(155.7)
```

prints 12.478.

Sample Program

```
68Ø INPUT "TOTAL RESISTANCE (OHMS)"; R
69Ø INPUT "TOTAL REACTANCE (OHMS)"; X
7ØØ Z = SQR((R * R) + (X * X))
71Ø PRINT "TOTAL IMPEDANCE (OHMS) IS" Z
```

This program computes the total impedance for series circuits.

STOP Statement

STOP

Stops program execution.

When a program encounters a STOP statement, it prints the message BREAK IN, followed by the line number that contains the STOP. STOP is primarily a debugging tool. During the break in execution, you can examine variables or change their values.

The CONT command resumes execution at the point it was halted. But if the program itself is altered during the break, CONT cannot be used.

Sample Program

```
226Ø X = RND(1Ø)
227Ø STOP
228Ø GOTO 226Ø
```

A random number between 1 and 1Ø is assigned to X, then program execution halts at line 227Ø. You can now examine the value X with PRINT X. Type CONT to start the cycle again.

STR$ Function

STR$(number)

Converts number into a string.

If number is positive, STR$ places a blank before the string.

While arithmetic operations may be performed on number, only string functions and operations may be performed on the string.

Example

```
S$ = STR$(X)
```

converts the number X into a string and stores it in S$.

Sample Program

```
5   REM ARITHMETIC FOR KIDS
1Ø  INPUT "TYPE A NUMBER"; N
15  ON LEN(STR$(N)) GOSUB 3Ø, 1ØØ, 2ØØ, 3ØØ, 4ØØ, 5ØØ
```

STRING$ Function

STRING$(number, character)

Returns a string of number characters.

Number must be in the range Ø to 32767.

Character is a string or an ASCII code. If you use a string constant, it must be enclosed in quotes.

All the characters in the string will have either the ASCII code specified, or the first letter of the string specified.

STRING$ is useful for creating graphs or tables.

Examples:

```
B$ = STRING$(25, "X")
```

puts a string of 25 "X"s into B$.

```
PRINT STRING$(5Ø, 1Ø)
```

prints 5Ø blank lines on the display, since 1Ø is the ASCII code for a line feed.

Sample Program

```
1Ø4Ø CLEAR 3ØØ
1Ø5Ø INPUT "TYPE IN THREE NUMBERS BETWEEN 33 AND
     159(N1,N2,N3)"; N1, N2, N3
1Ø6Ø CLS: FOR I = 1 TO 4: PRINT STRING$(2Ø, N1): NEXT I
1Ø7Ø FOR J = 1 TO 2: PRINT STRING$(4Ø, N2): NEXT J
1Ø8Ø PRINT STRING$(8Ø, N3)
```

SWAP Statement

SWAP variable1, variable2

Exchanges the values of two variables.

Variables of any type may be SWAPed (integer, single precision, double precision, string). However, both must be of the same type, otherwise, a "Type mismatch" error results.

Either or both of the variables may be elements of arrays. If one or both of the variables are non-array variables which have not been assigned values, an "Illegal Function Call" error results.

Example

```
SWAP F1#, F2#
```

swaps the contents of F1# and F2#. The contents of F2# are put into F1#, and the contents of F1# are put into F2#.

Sample Program

```
1Ø A$="ONE":B$="ALL":C$="FOR"
2Ø PRINT A$ C$ B$
3Ø SWAP A$, B$
4Ø PRINT A$ C$ B$
RUN
ONEFORALL
ALLFORONE
```

SYSTEM Statement

SYSTEM

Closes all files and exits MBASIC. The resident MBASIC program will be lost.

Example

```
SYSTEM
```

returns you to TRS-XENIX level.

When SYSTEM is executed it spools the LPRINT buffer to the line printer.

TAB Function

TAB (number)

Spaces to position number on the display.

Number must be in the range 1 to 32767.

If the current print position is already beyond space number, TAB goes to that position on the next line. Space one is the leftmost position; the width minus one is the rightmost position.

TAB may only be used with the PRINT and LPRINT statements.

Sample Program

```
10 PRINT "NAME" TAB(25) "AMOUNT":PRINT
20 READ A$, B$
30 PRINT A$ TAB(25) B$
40 DATA "G.T.JONES","$25.00"
RUN
```

The display shows:

```
NAME                        AMOUNT

G.T.JONES                   $25.00
```

TAN Function

TAN(number**)**

Computes the tangent of number.

Number must be in radians. To obtain the tangent of number when it is in degrees, use TAN (number * .11745329). The result is always double precision.

Examples

```
PRINT TAN(7.96)
```

prints -9.396962Ø13Ø868

Sample Program

```
72Ø INPUT "ANGLE IN DEGREES"; ANGLE
73Ø T = TAN(ANGLE * .Ø1745329)
74Ø PRINT "TAN IS" T
```

TIME$ Function

TIME$

Returns the time of the day.

This function lets you use the time in a program.

The operator sets the time initially when TRS-XENIX is started up. When you request the time, TIME$ supplies it using this format:

14:47:18

which means 14 hours, 47 minutes and 18 seconds (24-hour clock).

Example

```
A$ = TIME$
```

stores the current time in A$.

Sample Program

```
114Ø IF LEFT$(TIME$, 5) = "1Ø:15" THEN PRINT "Time is
     1Ø:15 A.M.--time to pick up the mail." : END
115Ø GOTO 114Ø
```

TROFF, TRON Function

TROFF
TRON

Turn the "trace function" on/off.

The trace function lets you follow program flow. This is helpful for debugging and analyzing of the execution of a program.

Each time the program advances to a new line, TRON displays that line number inside a pair of brackets. TROFF turns the tracer off.

Sample Program

```
229Ø TRON
23ØØ X = X * 3.14159
231Ø TROFF
```

Lines 229Ø and 231Ø above might be helpful in assuring you that line 23ØØ is actually being executed, since each time it is executed [23ØØ] is printed on the Display.

After a program is debugged, the TRON and TROFF statements can be removed.

UNLOCK Statement

UNLOCK [#]buffer [range]

Releases access restrictions placed upon a file.

UNLOCK releases locks in the specified range on the file opened with the number buffer.

Range is specified only if the file is open for random access. The syntax for range is record1 TO record2.

Sample Program

```
10 OPEN "R", 1, "employee.data", 32
20 INPUT "Record number"; N%
30 If N%>200 THEN 110
40 LOCK 1, N%
50 GET 1, N%
60 PRINT NAME$
70 PRINT ADDRESS$
80 PRINT TEL$
90 UNLOCK 1, N%
100 GOTO 30
110 CLOSE 1
```

This program opens the file **employee.data** and locks the records requested. As the information of each record requested is written out, the corresponding record is unlocked. This process continues until you request a record number larger than 200.

VAL Function

VAL(string)

Calculates the numerical value of string.

VAL is the inverse of the STR$ function; it returns the number represented by the characters in a string argument. This number may be integer, single precision, or double precision, depending on the range of values and the rules used for typing all constants.

For example, if A$ = "12" and B$ = "34" then VAL(A$ + "." + B$) returns the value 12.34 and VAL(A$ + "E" + B$) returns the value 12E34, that is, 12 * 10^34.

VAL terminates its evaluation on the first character which has no meaning in a numeric term.

If the string is non-numeric or null, VAL returns a zero.

Examples

```
PRINT VAL("100 DOLLARS")
```

prints 100.

```
PRINT VAL("1234E5")
```

prints 1.234E+08.

```
B = VAL("3" + "*" + "2")
```

assigns the value 3 to B (the asterisk has no meaning in a numeric term).

Sample Program

```
10  READ NAME$, CITY$, STATE$, ZIP$
20  IF VAL(ZIP$) < 90000 OR VAL(ZIP$) > 96699 THEN PRINT NAME$
    TAB(25) "OUT OF STATE"
30  IF VAL(ZIP$) > 90801 AND VAL(ZIP$) <= 90815 THEN PRINT
    NAME$ TAB(25) "LONG BEACH"
```

VARPTR Function

VARPTR(variable or buffer)

Returns the absolute memory address.

VARPTR can help you locate a value in memory. When used with variable, it returns the address of the first byte of data identified with variable.

When used with buffer, it returns the address of the file's data buffer. If the variable you specify has not been assigned a name, or the file has not been opened, an "Illegal Function Call" occurs.

If VARPTR (integer variable) returns address K:
Address K contains the most significant byte (MSB) of 2-byte integer.
Address K + 1 contains the least significant byte (LSB) of integer.

If VARPTR (single-precision variable) returns address K:

(K)* = exponent of value. High bit is sign bit. Bias is 64
(K) + 1 = MSB of BCD mantissa (2 digits)
(K) + 2 = next MSB (2 digits)
(K) + 3 = LSB (2 digits)

If VARPTR (double-precision variable) returns K:

(K)* = exponent of value. High bit is sign bit. Bias is 64
(K) + 1 = MSB of BCD mantissa (2 digits)
(K) + 2 = next MSB (2 digits)
(K) + 3 = next MSB (2 digits)
(K) + 4 = next MSB (2 digits)
(K) + 5 = next MSB (2 digits)
(K) + 6 = next MSB (2 digits)
(K) + 7 = LSB (2 digits)

Note: You cannot return VARPTR to an integer variable.

For single and double-precision values, the number is stored in normalized exponential form, so that a decimal is assumed before the MSB. 64 is added to the exponent. Furthermore, the high bit of MSB is used as a sign bit. It is set to Ø if the number is positive or to 1 if the number is negative. See examples below.

*(K) signifies "contents of address K"

If VARPTR (string variable) returns K:

(K)* = MSB length of string
(K + 1) = LSB length of string
(K + 2) = MSB of string value starting address
(K + 3) = next MSB of string value starting address
(K + 4) = LSB of string value starting address

The address will probably be in high RAM where string storage space has been set aside. But, if your string variable is a constant (a string literal), then it will point to the area of memory where the program line with the constant is stored, in the program buffer area. Thus, program statements like A$="HELLO" do not use string storage space.

VARPTR(array variable) returns the address for the first byte of that element in the array. The element consists of 2 bytes if it is an integer array; 5 bytes if it is a string array; 4 bytes if it is a single precision array; and 8 bytes if it is a double precision array.

The first element in the array is preceded by:
1. A sequence of two bytes per dimension, each two-byte pair indicating the "depth" of each respective dimension.
2. A single byte indicating the total number of dimensions in the array.
3. Three bytes indicating the total number of elements in the array.
4. A two-byte pair containing the ASCII-coded array name.
5. A one-byte type-descriptor(Ø2 = Integer, Ø5=String, Ø4 = Single=Precision, Ø8 = Double-Precision).

Item 1 immediately precedes the first element, Item 2 precedes Item 1, and so on.

The elements of the array are stored sequentially with the first dimension-subscripts varying "fastest", then the second, etc.

Examples

A! = 2 is stored as follows:

2 = 2X1Ø (to the first)

So exponent of A is 64+1 = 65 (called excess 64)

The high bit of the exponent is set to zero to indicate a positive number.

So A! = 2 is stored as:

```
(K)*      65
(K) + 1   32
(K) + 2    Ø
(K) + 3    Ø
```

A! = -5 is stored as:

```
(K)*     192
(K) + 1   8Ø
(K) + 2    Ø
(K) + 3    Ø
```

A! = 123.456 is stored as:

```
(K)*      67
(K) + 1   18
(K) + 2   52
(K) + 3   86
```

A! = -123.456 is stored as:

```
(K)*     195
(K) + 1   18
(K) + 2   52
(K) + 3   86
```

The following program will allow you to see the contents of a single precision variable.

```
1Ø INPUT "ENTER A NUMBER "; A!
2Ø B = VARPTR(A!)
3Ø FOR I = B TO B+3
4Ø PRINT PEEK(I)
5Ø NEXT I
6Ø END
RUN
ENTER A NUMBER? 1ØØ
67
16
Ø
Ø
Ok
```

To look at double precision use:

```
10 INPUT "ENTER A NUMBER "; A
20 B = VARPTR(A)
30 FOR I = B TO B+7
40 PRINT PEEK(I)
50 NEXT I
60 END
RUN
ENTER A NUMBER? 123456789
73
18
52
86
120
144
0
0
Ok
```

WHILE....WEND Statement

WHILE expression

.
.
.
.
{loop statements}
.
WEND

Execute a series of statements in a loop as long as a given condition is true.

If expression is not zero (true), MBASIC executes loop statements until it encounters a WEND. MBASIC returns to the expression. If it is still true, MBASIC repeats the process. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND matches the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND causes a "WEND without WHILE" error.

Sample Program

```
9Ø 'BUBBLE SORT ARRAY A$
1ØØ FLIPS=1 'FORCE ONE PASS THRU LOOP
11Ø WHILE FLIPS
115 FLIPS=Ø
12Ø FOR I=1 TO J-1  'J = NUMBER OF ELEMENTS IN A$()
13Ø IF A$(I)>A$(I+1)THEN SWAP A$(I), A$(I+1): FLIPS=1
14Ø NEXT I
15Ø WEND
```

This program sorts the elements in array A$. Control falls out of the WHILE loop when no more SWAPS are performed on line 13Ø.

WIDTH Statement

WIDTH buffer, size

or

WIDTH device, size

Sets the size for lines to be printed from a file which is OPENed for output.

Size must be an integer between 1 and 255. It sets the number of characters that can be printed on one line. If size is 255, the line width is "infinite", that is, MBASIC never inserts a carriage return

Device identifies a device such as "LPT1:" or "SCRN:". For more information on devices, see OPEN and Appendix G.

Buffer is the buffer used to OPEN the file.

WIDTH buffer, size changes the line width of the file associated with buffer. That is, if more than size bytes are output on one line to the file, MBASIC forces a new file.

WIDTH device, size has no effect on files currently OPENed to that device. A subsequent OPEN device FOR OUTPUT AS number will use size for width while the file is OPEN.

Sample Program

Suppose that line 1Ø is in a file which is accessed by buffer 1. The file is currently OPEN for output:

```
1Ø PRINT "DON QUIXOTE DE LA MANCHA"
RUN
DON QUIXOTE DE LA MANCHA
OK

WIDTH 1, 15
RUN
Ok

DON QUIXOTE DE
LA MANCHA
Ok
```

10/16/85 5136#1 to change width on screen format is WIDTH #X <- # of characters per line

WRITE Statement

WRITE [data,...]

Writes data on the display.

WRITE prints the values of the data items you type. If data is omitted, MBASIC prints a blank line. The data may be numeric and/or string. They must be separated by commas.

When the data is printed, each data item is separated from the last by a comma. Strings are delimited by quotation marks. After printing the last item on the list, MBASIC inserts a carriage return.

Example

```
10 A=95:B=76:C$="GOOD BYE"
20 WRITE A,B,C$
RUN
 95, 76, "GOOD BYE"
OK
```

WRITE# Statement

WRITE# buffer, data,...

Writes data to a sequential-access file.

Buffer must be the number used to OPEN the file.

The data you enter may be numeric or string expressions.

WRITE# inserts commas between the data items as they are written to disk. It delimits strings with quotation marks. Therefore, it is not necessary to put explicit delimiters between the data.

The items on data must be separated by commas.

WRITE# inserts a carriage return after writing the last data item to disk.

For example, if

A$="MICROCOMPUTER" and B$="NEWS"

the statement

WRITE#1, A$,B$

writes the following image to disk:

"MICROCOMPUTER", "NEWS"

Appendices

MBASIC Reference Manual

Appendix A

Error Messages

Number	Message
1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	Syntax error MBASIC encountered a line that contains an incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.). MBASIC automatically enters the edit mode at the line that caused the error.
3	Return without GOSUB MBASIC encountered a RETURN statement for which there is no matching GOSUB statement.
4	Out of data MBASIC encountered a READ statement but there are no DATA statements with unread items remaining in the program.
5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: 1. A negative or unreasonably large subscript. 2. A negative or zero argument with LOG. 3. A negative argument to SQU. 4. A negative mantissa with a noninteger exponent.

5. An improper argument to MID$, LEFT$, RIGHT$, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

6 Overflow

The result of a calculation is too large to be represented in MBASIC numeric format. If underflow occurs, the result is zero and execution continues without an error.

7 Out of memory

A program is too large, or has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

8 Undefined line number

A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.

9 Subscript out of range

An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts.

10 Duplicate Definition

Two DIM statements are given for the same array; or, a DIM statement is given for an array after the default dimension of 10 has been established for that array.

11 Division by zero

An expression includes division by zero, or the operation of involution results in zero being raised to a negative power. MBASIC supplies machine infinity with the sign of the numerator as the result of the division; or it supplies positive machine infinity as the result of the involution. Execution then continues.

12 Illegal direct

A statement that is illegal in direct mode is entered as a direct mode command.

13 Type mismatch

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

14 Out of string space

String variables have caused MBASIC to exceed the amount of free memory remaining. MBASIC allocates string space dynamically, until it runs out of memory.

15 String too long

An attempt is made to create a string more than 32767 characters long.

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

An attempt is made to continue a program that:

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

18 Undefined user function.

An attempt is made to call a function which has not been defined.

19 No RESUME

An error handling routine is entered but contains no RESUME statement.

2Ø	RESUME without error A RESUME statement is encountered before an error handling routine is entered.
21	Unprintable error An error message is not available for the error condition which exists.
22	Missing operand An expression contains an operator with no operand.
23	Line buffer overflow An attempt has been made to input a line that has too many characters.
26	FOR without NEXT A FOR statement was encountered without a matching NEXT.
29	WHILE without WEND A WHILE statement does not have a matching WEND.
3Ø	WEND without WHILE A WEND statement was encountered without a matching WHILE.
5Ø	Field overflow A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
51	Internal error An internal malfunction has occurred in MBASIC. Report to Radio Shack the conditions under which the message appeared.

52 Bad file number

A statement or command references a file with a file number (buffer number) that is not open or is out of the range of file numbers specified at initialization.

53 File not found

A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode

An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to execute an OPEN statement with a file mode other than I, O, R, A or D.

55 File already open

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

57 Device I/O error

an Input/Output error occurred. This is a fatal error; the operating system cannot recover it.

58 File already exists

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

61 Disk full

All disk storage space is in use.

62 Input past end

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

63 Bad record number

In a PUT or GET statement, the record number is either greater than the maximum allowed (32,767) or equal to zero.

64 Bad file name

An illegal form is used for the filename with a LOAD, SAVE, KILL, or OPEN statement (e.g., a filename with too many characters).

66 Direct statement in file

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

67 Too many files

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

68 Device Unavailable

An attempt is made to access a device which is not ready or unavaiable.

70 Permission Denied

An attempt is made to either: access a LOCKed file; OPEN a file for read without read permission or for write without write permission; LOAD a file without load permission, or SAVE a file without save permission.

73 Advanced Feature

An attempt is made to use a feature which has not been implemented yet.

77 Deadlock

TRS-XENIX MBASIC has detected a deadlock.

Appendices

Appendix B

Reserved Words

A reserved word with a dollar-sign ("$") after it may be used as a numeric variable name if the dollar-sign is dropped. For instance, CHR and CHR# are valid variable names. However, DEF statements may not be used to assign values to this type of variable.

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

Appendix C

Converting TRSDOS BASIC Programs to MBASIC

This Appendix is divided into three parts:

Part I	BASIC Files
Part II	Conversion Commands
Part III	Conversion Procedures

Please read all three sections for a full understanding of this Appendix.

Part I / BASIC Files

A BASIC file can be transferred from one of three sources: a TRSDOS-II (4.Ø or later) System diskette, a TRSDOS-II (4.Ø or later) SAVE diskette, or a TRSDOS 2.Øa System diskette. Each of these sources has a unique procedure for converting to TRS-XENIX.

There are two types of BASIC files: BASIC Program Files and BASIC Data Files.

BASIC Program Files. This file can be either compressed or ASCII format. Knowing the type of format is important when transferring files to TRS-XENIX.

Compressed Format. A BASIC program file stored in compressed format is fixed length (type F) and has a record length of 256 bytes. A compressed BASIC program is stored by using the BASIC SAVE command without the A option.

The compressed format of TRSDOS BASIC and TRS-XENIX MBASIC are different, therefore TRSDOS BASIC compressed format files must be converted to TRS-XENIX MBASIC ASCII format. This is explained in detail later.

TRSDOS-II 4.x System Diskette -- To transfer a compressed BASIC program file stored on a TRSDOS-II 4.x System diskette to TRS-XENIX, refer to Procedure 1 in Part III of this Appendix.

TRSDOS-II 4.x SAVE Diskette -- To transfer a compressed BASIC program stored in TRSDOS-II 4.x SAVE format to TRS-XENIX, refer to Procedure 2 in Part III of this Appendix.

TRSDOS 2.Øa System Diskette -- To transfer a compressed BASIC program stored on a TRSDOS 2.Øa System diskette to TRS-XENIX, refer to Procedure 3 in Part III of this Appendix.

ASCII Format. A BASIC program file in ASCII format is fixed length (type F) and has a record length of 1 byte. As ASCII BASIC program stored by using the BASIC SAVE command with the A option.

Note: If a TRSDOS ASCII BASIC program is transferred to TRS-XENIX, extensive editing may be required to properly pad MBASIC keywords to the correct syntax. If you have TRSDOS BASIC programs stored in the ASCII format, it is suggested that you save them again in compressed format. You can do this by loading your program into TRSDOS BASIC and resaving it without the A option.

TRSDOS-II 4.x System Diskette -- To transfer an ASCII BASIC program stored on a TRSDOS-II 4.x System diskette to TRS-XENIX, refer to Procedure 4 in Part III of this Appendix.

TRSDOS-II 4.x SAVE Diskette -- To transfer an ASCII BASIC program stored in TRSDOS-II 4.x SAVE format to TRS-XENIX, refer to Procedure 5 in Part III of this Appendix.

TRSDOS 2.Øa System Diskette -- To transfer an ASCII BASIC program stored on a TRSDOS 2.Øa System diskette to TRS-XENIX, refer to Procedure 6 in Part III of this Appendix.

BASIC Data Files. BASIC Data Files are files which have been created by BASIC programs. There are two types of BASIC Data Files, sequential access and random access.

Refer to Chapter 5 for more information on BASIC Data Files.

Sequential Access

TRSDOS-II 4.x System Diskette -- To transfer a sequential access data file stored on a TRSDOS-II 4.x System diskette to TRS-XENIX, refer to Procedure 7 in Part III of this Appendix.

TRSDOS-II 4.x SAVE Diskette -- To transfer a sequential access data file stored in TRSDOS-II 4.x SAVE format to TRS-XENIX, refer to Procedure 8 in Part III of this Appendix.

TRSDOS 2.Øa System Diskette -- To transfer a sequential access data file stored on a TRSDOS 2.Øa System diskette to TRS-XENIX, refer to Procedure 9 in Part III of this Appendix.

Direct Access

TRSDOS-II 4.x System Diskette -- To transfer a direct access data file stored on a TRSDOS-II 4.x System diskette to TRS-XENIX, refer to Procedure 1Ø in Part III of this Appendix.

TRSDOS-II 4.x SAVE Diskette -- To transfer a direct access data file stored in TRSDOS-II 4.x SAVE format to TRS-XENIX, refer to Procedure 11 in Part III of this Appendix.

TRSDOS 2.Øa System Diskette -- To transfer a direct data file stored on a TRSDOS 2.Øa System diskette to TRS-XENIX, refer to Procedure 12 in Part III of this Appendix.

Part II / Conversion Commands

The TRS-XENIX commands **tx** and **bp** are used to transfer TRSDOS-II BASIC files to TRS-XENIX. The **tx** command transfers a TRSDOS-II file to TRS-XENIX. The **bp** command then converts the transferred BASIC compressed format program file into an ASCII format that MBASIC can read and execute.

The syntax for **tx** is:

tx :d options Tfspec Xfspec.braw

Refer to the **tx** command in your TRS-XENIX Operations Guide for an explanation of options.

The syntax for **bp** is:

bp Xfspec.braw >Xfspec.bas

Where:
- Tfspec = TRSDOS-II file specification
- Xfspec.braw = TRS-XENIX file specification containing TRSDOS-II BASIC compressed format
- Xfspec.bas = TRS-XENIX destination file specification
- :d = specifies the drive that contains the TRSDOS-II diskette. :d is required.

The **FCOPY** command copies disk files that were created and stored on a TRSDOS 2.Øa System diskette to a disk formatted by TRSDOS-16 or TRSDOS-II and vice versa. You must **FCOPY** any disk files created with TRSDOS before you can use them with the conversion command **tx.**

The syntax for **FCOPY** is:

FCOPY Tfspec **TO** T-IIfspec [options]

Where: Tfspec = TRSDOS file specification
T-IIfspec = TRSDOS-II file specification

Refer to your TRSDOS-II or TRSDOS-16 owner's manual for an explanation of options.

Important Notes:

1. For information on the size of variables allowed and syntax padding requirements, refer to Appendices D and H.

2. TRSDOS BASIC programs designed to use Z8Ø machine language subroutines cannot run successful without the subroutines being rewritten in 68ØØØ machine language and the BASIC source changed from using the DEF USR and USR commands to the CALL command. If you desire to use the machine language subroutines with MBASIC, you need the TRS-XENIX Development System.

Part III / Conversion Procedures

1. This procedure transfers the TRSDOS-II BASIC compressed program file PROGRAM/BAS stored on a TRSDOS-II 4.x System diskette in Drive Ø, to TRS-XENIX.

```
tx :Ø -va PROGRAM/BAS program.braw
```

transfers the compressed format of PROGRAM/BAS to TRS-XENIX and stores the file in the temporary program file **program.braw.**

```
bp program.braw >program.bas
```

converts **program.braw** into ASCII format and stores the file in **program.bas.**

```
rm program.braw
```

removes the temporary program file **program.braw.**

2. This procedure transfers the TRSDOS-II BASIC compressed program file SPROGRAM/BAS stored on your TRSDOS-II 4.x SAVE diskette in Drive Ø, to TRS-XENIX.

```
tx :Ø -rva SPROGRAM/BAS sprogram.braw
```

transfers the compressed format of SPROGRAM/BAS to the TRS-XENIX and stores the file in the temporary program file **sprogram.braw.**

```
bp sprogram.braw >sprogram.bas
```

converts **sprogram.braw** into ASCII format and stores the file in **sprogram.bas.**

```
rm sprogram.braw
```

removes the temporary program file **sprogram.braw.**

3. This procedure transfers the TRSDOS BASIC compressed program file TPROGRAM/BAS, stored on your TRSDOS 2.Øa System diskette in Drive Ø, to TRS-XENIX.

Boot up your system with a TRSDOS-II 4.x System diskette in Drive Ø to TRS-XENIX and your TRSDOS 2.Øa System diskette in Drive 1.

```
FCOPY TPROGRAM/BAS:1 TO PROGRAM/BAS:Ø
```

copies TPROGRAM/BAS to your TRSDOS-II diskette and renames it as PROGRAM/BAS.

Reboot your system under the TRS-XENIX with your 4.x System diskette in Drive Ø.

```
tx :Ø -va PROGRAM/BAS program.braw
```

transfers the compressed format of PROGRAM/BAS to the TRS-XENIX and stores the file in the temporary program file **program.braw.**

```
bp program.braw >program.bas
```

converts **program.braw** into ASCII format and stores the file in **program.bas.**

```
rm program.braw
```

removes the temporary program file **program.braw.**

4. This procedure transfers the TRSDOS-II ASCII program file PROGRAM/TXT, stored on your 4.x System diskette in Drive Ø, to TRS-XENIX.

```
tx :Ø -vax PROGRAM/TXT program.bas
```

transfers PROGRAM/TXT to TRS-XENIX and stores it in **program.bas.**

5. This procedure transfers the TRSDOS-II ASCII program file SPROGRAM/TXT, stored on your TRSDOS-II 4.x SAVE diskette in Drive Ø, to TRS-XENIX.

```
tx :Ø -vaxr SPROGRAM/TXT sprogram.bas
```

transfers SPROGRAM/TXT to TRS-XENIX and stores it in **sprogram.bas.**

6. This procedure transfers the TRSDOS ASCII program file TPROGRAM/TXT, stored on your TRSDOS 2.Øa System diskette in Drive Ø, to TRS-XENIX.

Boot your system with a TRSDOS-II 4.x System diskette in Drive Ø and your TRSDOS 2.Øa System diskette in Drive 1..

```
FCOPY TPROGRAM/TXT:1 TO PROGRAM/TXT:Ø
```

copies TPROGRAM/TXT to your TRSDOS-II diskette and stores it in PROGRAM/TXT.

Reboot your system under TRS-XENIX with your TRSDOS-II 4.x System diskette in Drive Ø.

```
tx :Ø -vax PROGRAM/TXT program.bas
```

transfers PROGRAM/TXT to TRS-XENIX and stores it in **program.bas.**

7. This procedure transfers the TRSDOS-II sequential data file PROGRAM/DAT, stored on your TRSDOS-II 4.x System diskette in Drive Ø, to TRS-XENIX.

tx :Ø -vax PROGRAM/DAT program.dat

transfers PROGRAM/DAT to TRS-XENIX and stores it in **program.dat**

8. This procedure transfers the TRSDOS-II sequential data file SPROGRAM/DAT, stored on your TRSDOS-II 4.x SAVE diskette in Drive Ø, to TRS-XENIX.

   tx :Ø -vaxr SPROGRAM/DAT sprogram.dat

   transfers SPROGRAM/DAT to TRS-XENIX and stores it in **sprogram.dat.**

9. This procedure transfers the TRSDOS sequential data file TPROGRAM/DAT, stored on your TRSDOS 2.Øa System diskette in Drive Ø, to TRS-XENIX.

   Boot up your system with a TRSDOS-II 4.x System diskette in Drive Ø and your TRSDOS 2.Øa System diskette in Drive 1.

   FCOPY TPROGRAM/DAT:1 TO PROGRAM/DAT:Ø

   copies TPROGRAM/DAT to your TRSDOS-II diskette and stores it in PROGRAM/DAT.

   Reboot your system under TRS-XENIX with your TRSDOS-II 4.x diskette in Drive Ø.

   tx :Ø -vax PROGRAM/DAT program.dat

   transfers PROGRAM/DAT to TRS-XENIX and stores it in **program.dat.**

1Ø. This procedure transfers the TRSDOS-II random data file PROGRAM/RDT, stored on your TRSDOS-II 4.x System diskette in Drive Ø, to TRS-XENIX.

   tx :Ø -va PROGRAM/RDT program.rdt

   transfers PROGRAM/RDT to TRS-XENIX and stores it in **program.rdt.**

11. This procedure transfers the TRSDOS-II random data file SPROGRAM/RDT, stored on your TRSDOS-II 4.x SAVE diskette in Drive Ø, to TRS-XENIX.

```
tx :Ø -var SPROGRAM/RDT sprogram.rdt
```

transfers SPROGRAM/RDT to TRS-XENIX and stores it in **sprogram.rdt.**

12. This procedure transfers the TRSDOS random data file TPROGRAM/RDT, stored on your TRSDOS 2.Øa System diskette in Drive Ø, to TRS-XENIX.

    Boot up your system with a TRSDOS-II 4.x System diskette in Drive Ø and your TRSDOS 2.Øa System diskette in Drive 1.

    ```
    FCOPY TPROGRAM/RDT:1 TO PROGRAM/RDT:Ø
    ```

    Copies TPROGRAM/RDT to your TRSDOS-II diskette and stores it in PROGRAM/RDT

    Reboot your system under TRS-XENIX with your TRSDOS-II 4.x System diskette in Drive Ø.

    ```
    tx :Ø -va PROGRAM/RDT program.rdt
    ```

    transfers PROGRAM/RDT to TRS-XENIX and stores it in **program.rdt.**

Appendix D

Differences Between TRSDOS BASIC and TRS-XENIX MBASIC

TRS-XENIX offers several enhancements over TRSDOS BASIC. The two most important ones are: It can be accessed by multiple users and it has more memory capacity. This appendix discusses these enhancements and other differences between the two BASICs.

1. Keywords. The definitions and/or syntaxes of certain keywords vary between TRSDOS BASIC and TRS-XENIX MBASIC. See: CLEAR, EOF, LIST, LLIST, LOC, LOF, LPRINT, OPEN, POS, SYSTEM, VARPTR, and WIDTH in Chapter 7.

   TRS-XENIX MBASIC does not support the following TRSDOS BASIC keywords: ROW, INP, OUT, ERR$, DEFUSR, USR.

   TRSDOS BASIC does not support the following TRS-XENIX MBASIC keywords: LOCK, LPOS, PEEK, POKE, SHELL, UNLOCK, WHILE..WEND, WIDTH, WRITE, and WRITE#. For a detailed discussion of these new keywords, see Chapter 7.

•2. Error Messages and Codes. The codes for TRS-XENIX MBASIC are different than those for TRSDOS BASIC. Refer to Appendix A for a list of error codes and messages for TRS-XENIX MBASIC.

•3. Precision of Variables. TRS-XENIX MBASIC assumes all your variables are double precision, unless you specify otherwise. TRSDOS BASIC assumes your variables are single precision.

4. Loading MBASIC. Chapter 1 of this manual explains how to load TRS-XENIX MBASIC.

5. Filespecs. Under TRS-XENIX MBASIC, filespecs may be as long as the largest legal string variable, that is, 32767 characters. Under TRSDOS BASIC, filespecs can be up to eight characters long.

6. Files. TRS-XENIX MBASIC allows 17 files to be OPEN at one time; TRSDOS BASIC allows only 15.

7. Memory Space. Your program, variables and string space (combined) may total up to 16777215 bytes in TRS-XENIX, compared to 33608 bytes in TRSDOS BASIC. (Remember, however, that your computer's memory capacity also limits the amount of memory MBASIC can use. The 16777215 bytes are only available if you have a 16-megabyte computer).

1 LOF in XENIX = # of bytes in record vs TRSDOS = last record # 01/07/84 dmm

8. Strings. In TRS-XENIX, string variables and string expressions may contain up to 32767 characters each. In TRSDOS, they may only contain up to 255 characters each.

9. Devices. With TRS-XENIX, you can access "devices" other than disk files. To access these devices, use the same syntax as for accessing files. For more information, see Appendix C and OPEN.

10. Assembly-Language Subroutines. TRSDOS BASIC operates under the Z80 microprocessor; TRS-XENIX operates under the 68000 processor. If your program has assembly-language subroutines, these subroutines need to be compatible with the 68000 processor. **Do not use any subroutines until you have re-written them for the 68000 processor.** See Appendix F, Loading Assembly-Language Files, for more information.

TRS-XENIX in a Multi-User Environment:

1. ASCII Codes. If your program uses ASCII codes, make sure they represent only text characters (codes 32 to 127). ASCII codes for video control characters, graphics characters and special characters may vary between different types of terminals. Therefore, each terminal may interpret a single ASCII code in a different way. TRS-XENIX MBASIC can clear the screen (through the CLS statement) and position the cursor (through the TAB function) in a multi-user environment. However, it does not support other video control functions, such as switching to double-sized characters, reverse video, etc.

•2. Files. If more than one user is accessing a single disk file at the same time, unexpected results may occur (such as loss of a file's contents). To avoid this, use the LOCK and UNLOCK commands. It is important to understand these commands before attempting to access files in a multi-user environment.

Appendix E

Machine Language Interface to TRS-XENIX MBASIC

The CALL statement supports standard "C" calling conventions. When executing the CALL statement, all parameters must be variables. Also, no previously unreferenced scalar variable may follow an array element in the parameter list. If you attempt this, an "Illegal function call" error is generated.

The called routine has the following environment upon entry:
Stack Pointer-->Return Address (32 bits)--low memory
Pointer to value of 1st parameter... (32 bits)
Pointer to value of last parameter (32 bits)--high memory

On exit, parameters are left on the stack (MBASIC will remove them). The called routine may destroy registers aØ, a1, dØ, and d1 but all other registers must be preserved.

Appendix F

Loading Assembly Language Files

MBASIC for the 68ØØØ provides a TRS-XENIX integrated mechanism for loading assembly language programs. To link and load assembly language files with MBASIC, start by using the -l ("el") option with the MBASIC command line. The file can be any relocatable file (either a.out, b.out, or x.out that has not been processed with ld utility without the -r option. See the TRS-XENIX manual A.OUT(5)). Only one file per -l is allowed; if more than one file needs to be loaded, use additional -l parameters. For example: If the assembly language files were asm1.s and asm2.s a possible syntax to assemble and run MBASIC with these assembly language files would be:

```
as -o asm1.o asm1.s
as -o asm2.o asm2.s
mbasic -l asm1.o -l asm2.o
```

When the -l option is used, the relocatable file /usr/bin/b68k.o is loaded with all the specified assembly language files and the "C" library, and a local file named b68k.o is produced and executed. If no -l option appears in the command line, the file /usr/bin/b68k is executed. (/usr/bin/b68k is the result of having loaded /usr/bin/b68k.o with the "C" library.)

It is important to understand that a local version of MBASIC is the executing program when the -l option is used with mbasic command. By local it is meant that a copy of b68k now resides in the user's current directory. This version of MBASIC contains the assembly language programs not the version of MBASIC residing in /bin. A new copy of MBASIC will be created in the current directory each time mbasic is executed with the -l option. If the same set of assembly language routines are to be used again the user might wish to execute the local version of MBASIC called b68k that was created the first time mbasic was executed with the -l option. The user can execute this local version of MBASIC by specifying the local file name. This would be the pathname to the current directory followed by b68k. Often the user profile is set to search for local copies of a file first. If this is the case then specifying b68k will surfice.

To sum up, MBASIC under TRS-XENIX on the 68ØØØ creates a special version of MBASIC called b68k when machine level routines are to be interfaced with MBASIC. If these machine level routines are to be called during MBASIC program execution this special version of

MBASIC must be executing. It will be executing as a result of using the -l option with mbasic TRS-XENIX command or by executing a local version of MBASIC directly after it has been created by an initial use of the mbasic command.

Appendix G

Device Handling

TRS-XENIX Devices

TRS-XENIX MBASIC supports Generalized Device I/O. This means that you can access "devices" other than disk files by using the same syntax which has always been used by MBASIC to access disk files. TRS-XENIX MBASIC supports the following "devices":

The Screen Device

The device "SCRN:" allows files to be opened to your computer screen for output. All data written to a file which is opened to "SCRN:" is directed to the standard output device (Screen). For example:

```
OPEN "SCRN:" for OUTPUT as #1
PRINT#1, N$
```

allows you to output to your computer screen and writes the value of N$ into the file assigned to buffer number 1, which is your computer screen.

The Keyboard Device

The device "KYBD:" allows files to be opened to your computer keyboard for input. All data read from a file which is opened to "KYBD:" comes from the standard input device (Keyboard). For example:

```
OPEN "KYBD:" for INPUT as #1
INPUT#1, N$
```

allows you to input from your keyboard and inputs the value of N$ into your program from the file assigned to buffer number 1, which is your keyboard.

The Printer Device

The device "LPT1:" allows files to be opened to your printer for output. All data written to a file which is opened to "LPT1:" is directed to the line printer. For example:

```
OPEN "LPT1:" for OUTPUT as #1
WRITE#1, S$
```

opens an output file to your line printer and writes the value of S$ to the file assigned.

TRS-XENIX pipes

TRS-XENIX supports pipes, which transfer the output of one program into the input of another program. Opening a file to device "pipe:.." opens a pipe, forks, and executes the specified child process. For example:

```
LIST "PIPE: lpr"
```

generates a listing to the lpr spooler.

```
OPEN "pipe:ls -l" for INPUT AS #1
```

allows the directory to be read via buffer number 1.

```
LIST 5Ø, "PIPE:write david"
```

lists line 5Ø on **"david's"** terminal using the write utility.

Line Printer Support

The command **LLIST** line number-line number is synonymous with the command **LIST** line number-line number, **"LPT1:"**. Both commands cause a listing to be sent to the line printer spooler. If <CTRL C> is pressed during either of these commands, the output is discarded.

LPRINT causes an invisible file number to be opened to a line printer spooler. This spooled output is released by the statements END, SYSTEM, and CLEAR. If CLEAR is used without parameters it does not release spooled LPRINT output.

If two files are opened to "LPT1:", their output is not intermixed, but spooled separately, nor is their output intermixed with LPRINT output.

Appendix H

Decimal Math Representation

The following describes the internal representation of numbers in TRS-XENIX MBASIC:

Double precision contains eight bytes as follows: One bit sign followed by 7 bits of biassed exponent followed by fourteen digits of mantissa, 4 bits each. If the sign bit is 1, the number is negative. The unbiassed exponent (biassed exponent - 64) is the power of 10 that the mantissa is to be multiplied by. The mantissa represents a number between 0.1000000000000000 and 0.99999999999999. For example -.00000123456789 would be represented by the hexadecimal number BB123456789000000.

External Representation	Internal Representation
-9.9999999999999D+62	FF99999999999999
-1D-64	8110000000000000
0	00xxxxxxxxxxxxx
1D-64	0110000000000000
9.9999999999999D+62	7F99999999999999

Single precision contains four bytes as follows: One bit sign followed by 7 bits of biassed exponent followed by six digits of mantissa, 4 bits each. If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative. The unbiassed exponent (biassed exponent - 64) is the power of 10 that the mantissa is to be multiplied by. The mantissa represents a number between 0.100000 and 0.999999. For example, 15.6 would be represented by the hexadecimal number 42156000.

External Representation	Internal Representation
-9.99999E+62	FF999999
-1E-64	81100000
0	00xxxxxx
1E-64	01100000
9.99999E+62	7F999999

Integers are represented by a 16 bit 2's complement signed binary number. Integer math a is identical to that of Binary Math.

External Representation	Internal Representation
-32768	8ØØØ
-1	FFFF
Ø	ØØØØ
1	ØØØ1
32767	7FFF

INDEX

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG ROAD MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

02/83-SP

Printed in U.S.A.