

baZic^(R)

II

A High-level Interpretive Computer Language

Developed for Use With

MicroDoZ^(R)

and

CP/M^(R)

developed by

Micro Mike's, Inc.

Box 1440

Amarillo, Texas 79105 USA

revision 03/03

July 18, 1981

making technology uncomplicated ... for People

Copyright 1981 by Micro Mike's, Inc.

Unauthorized copying of this software and/or manual is illegal.

TABLE OF CONTENTS

i ADDENDUM -- HOW TO CONFIGURE baZic FOR CP/M FOR YOUR SYSTEM

1 INTRODUCTION 1

 1.1 Definition of Mnemonics 2

 1.2 Definition of Terms 3

2 DIRECT COMMANDS 6

 2.1 Programming Commands 6

 2.1.1 LIST a program 6

 2.1.2 DElete line numbers 7

 2.1.3 SCRatch a program 7

 2.1.4 RENumber a program 8

 2.1.5 AUTO line numbering 8

 2.1.6 Program SIZE 9

 2.1.7 SET MEMory upper limits 9

 2.2 Disk Commands 9

 2.2.1 CATalog a disk 10

 2.2.2 SAVE a program 11

 2.2.3 SAVE a New program 12

 2.2.4 LOAD a program 12

 2.2.5 BYE to disk operating system 13

 2.3 Execution Commands 13

 2.3.1 RUN a program 13

 2.3.2 CONTInue a program 14

 2.3.3 Execution Statements 14

3 STATEMENTS 15

 3.1 Program Data Statements 15

 3.1.1 DATA constants 15

 3.1.2 READ data constants 16

 3.1.3 RESTORE data pointer 17

 3.2 Input and Output Statements 17

 3.2.1 PRINT a variable 17

 3.2.1.1 Formatted Printing 18

 3.2.1.2 PRINT@ and !@ 20

 3.2.2 INPUT a variable 21

3.2.3	INPUT1 a variable	22
3.2.4	OUTput a byte	22
3.3	Branching Statements	22
3.3.1	GOTO a line number	22
3.3.2	GOSUBroutine	23
3.3.3	RETURN from a subroutine	23
3.3.4	ON value GOTO a line number	24
3.3.5	ON value GOSUBroutine.	24
3.3.6	IF THEN ELSE	24
3.3.7	FOR NEXT STEP EXIT	25
3.4	File Statements	27
3.4.1	CREATE a file	27
3.4.2	DESTROY a file	28
3.4.3	OPEN a channel	28
3.4.4	CLOSE a channel	29
3.4.5	READ a file variable	29
3.4.6	WRITE a file variable	30
3.4.7	APPEND a program	30
3.4.8	CHAIN to a program	31
3.4.9	DOSCoMmand	31
3.5	Miscellaneous Statements	32
3.5.1	DIMension a variable	32
3.5.2	REMark a comment	33
3.5.3	LET (assign a variable)	33
3.5.4	CLeAr the Screen	34
3.5.5	FILL a memory location	34
3.5.6	SET ERRor trapping	34
3.5.7	LINE length	35
3.5.8	STOP a program	35
3.5.9	END a program	36
4	BUILT-IN FUNCTIONS	37
4.1	Math Functions	37
4.1.1	ABSolute value	37
4.1.2	SiGN of a number	38
4.1.3	INTeger value	38
4.1.4	LOGarithmic value	38
4.1.5	EXPonential value	38
4.1.6	SQuare ROOT	39
4.1.7	SINe	39
4.1.8	COSine	39
4.1.9	ArCTaNgent	40

4.2	String Functions	40
4.2.1	LENGth of a string	40
4.2.2	CHaRacter\$	41
4.2.3	ASCii value	41
4.2.4	VALue	41
4.2.5	STRing\$	42
4.3	Input Functions	42
4.3.1	INput a CHaRacter\$	42
4.3.2	INput a byte	42
4.3.3	INput the STATus	43
4.3.4	OUTSTATus	43
4.4	File Functions	43
4.4.1	TYPe of file pointer	43
4.4.2	FILE type	44
4.4.3	FILESIZE	44
4.4.4	FILEPoiNtEr position	44
4.5	Miscellaneous Functions	45
4.5.1	RaNDom	45
4.5.2	EXAMine memory	45
4.5.3	FREE memory	46
4.5.4	TABulate	46
4.5.5	CALL machine language	46
4.5.6	ADDRes of a variable	47
5	OPERATORS	48
5.1	Arithmetic Operators	48
5.2	Relational Operators	48
5.3	Boolean Operators	49
5.4	Order of Evaluation	50
6	USER-DEFINED FUNCTIONS	51
6.1	DEFine a function	52
6.2	RETURN from a function	52
6.3	FuNctionEND	52
7	ERROR MESSAGES	53
7.1	Trappable Errors	53
7.1.1	ARGument (Error 1)	53
7.1.2	DIMENSION (Error 2)	54
7.1.3	OUT OF BOUNDS (Error 3)	54
7.1.4	TYPe (Error 4)	56
7.1.5	FORMAT (Error 5)	57
7.1.6	LINE NUMBER (Error 6)	57
7.1.7	FILE (Error 7)	58

7.1.8	HARD DISK (Error 8)	59
7.1.9	DIVIDE by ZERO (Error 9)	59
7.1.10	SYNTAX (Error 10)	59
7.1.11	READ (Error 11)	61
7.1.12	INPUT (Error 12)	61
7.1.13	ARGument MISMATCH (Error 13)	61
7.1.14	NUMERIC Overflow (Error 14)	61
7.1.15	STOP/control C (Error 15)	61
7.1.16	LENGTH (Error 16)	62
7.2	Non-trappable Errors	62
7.2.1	CONTINUE	62
7.2.2	CONTROL STACK	62
7.2.3	DOUBLE DEFinition	62
7.2.4	FUNCTION DEFinition	63
7.2.5	ILLEGAL DIRECT	63
7.2.6	INTERNAL STACK Overflow	63
7.2.7	MEMORY FULL	63
7.2.8	MISSING NEXT	63
7.2.9	NO PROGRAM	63
7.2.10	TOO LARGE OR NO PROGRAM	63
8	MISCELLANEOUS TOPICS	65
8.1	Line Editor	65
8.1.1	Control G	66
8.1.2	Control N	66
8.1.3	Control A	66
8.1.4	Control Q	67
8.1.5	Control Z	67
8.1.6	Control D	67
8.1.7	Control Y	67
8.2	Data Files	68
8.3	BASIC Differences	70
8.3.1	Strings	70
8.3.2	IF THEN Evaluation	71
8.3.3	Miscellaneous	71
9	UTILITY PROGRAMS	72
9.1	SHORTB	72
9.2	XREF	73
9.3	ICOPY	76
9.4	COPYFILE	80
9.5	COMPACT	83
9.6	Interfacing COPY Programs to baZic programs	84
9.7	CRT	85
9.8	Other CRTs	87
10	OPTIMIZING baZic PROGRAMS	88

11 PARTIAL SOURCE LISTING 89

 11.1 MicroDoZbaZic 89

 11.2 baZic for CP/M 90

APPENDIX A (ASCII TABLE) 92

APPENDIX B (OPCODE CHART) 96

ERRATA 1 97

ERRATA 2 98

ERRATA 3 99

How to Configure baZic for CP/M for Your System

baZic for CP/M distributed on 5-1/4" NorthStar disk is recorded in Format 16, or the standard CP/M 2.2 double density Lifeboat format for NorthStar Horizon or NorthStar Advantage.

baZic for CP/M distributed on 8" disk is recorded in single density, soft-sectored CP/M format, which virtually all machines using 8" floppies and the CP/M operating system can read.

Make a copy of the baZic distribution disk and file the master disk in a safe place. The copy of the disk should be write-enabled. Boot up CP/M 2.0 or a more recent version. The CP/M operating system prompt will be displayed as follows:

A>

Setting Upper Memory Limits and Turnkeying 10-digit baZic

baZic for CP/M, Release 04/03, finds its own MEMSET level at boot-up time. Releases 03/03 and earlier require the following MEMSET procedures:

In the following example, we will turnkey and set upper memory limits for 10-digit software floating point baZic. We will load 10-digit software floating point baZic into memory, toggle on the turnkey byte (271), enter a one-line turnkey program to be saved as a part of baZic, set baZic upper memory limits for 56K CP/M, exit baZic to CP/M and save the file "BAZIC10.COM," 54 blocks long, on the disk to preserve these changes.

Common MEMSET values for baZic for CP/M:

48K -- 42245	60K -- 54533
56K -- 50437	64K -- 58373

The MEMSET value of baZic may be different for your hardware or version of CP/M. You can use trial and error until you reach the proper MEMSET level or you can enter the following (the underlined responses are yours):

```
READY
!EXAM(7)*256+EXAM(6)-1 <CR>
```

The maximum MEMSET level should be printed on the screen. To set the upper memory limits the syntax is as in the following example (your response is underlined):

```
MEMSET 50437 <CR>
```

Either Sysgen the copy of the baZic disk or remove the CP/M boot disk from Drive A and place the copy of the baZic disk in Drive A and then do a warm boot (press CTRL and C keys simultaneously). The copy of the baZic distribution disk must be in Drive A to continue with the configuration.

Following is an example configuration of 10-digit software floating point baZic, 56K CP/M (your responses are underlined):

```
A>>^C
A>>BAZIC10 <CR>
READY
FILL 271,0 <CR>
READY
10 CHAIN "MENU" <CR>
MEMSET 50437 <CR>
READY
BYE <CR>
A>>SAVE 54 BAZIC10.COM <CR>
A>
```

baZic includes built-in cursor-addressing and clear screen commands/statements. If you want to run only existing NorthStar BASIC programs under baZic you do not need to use these features. So that baZic users with a variety of CRTs can use these cursor addressing and clear screen features, the baZic distribution disk includes a CRT configuration program, "CRT.002." In the following example we will configure 10-digit software floating point baZic for a SOROC IQ 120 terminal (your responses are underlined):

```
A>>REN BAZIC10.001=BAZIC10.COM <CR>
A>>BAZIC08 <CR>
READY
LOAD CRT <CR>
READY
RUN <CR>
```

The program "CRT.002" will be executed. You will now see the following on the screen:

```
THIS PROGRAM IS DESIGNED TO CONFIGURE BAZIC FOR A PARTICULAR CRT.
FOR THE PRINT@ AND THE CLS STATEMENTS TO WORK PROPERLY, BAZIC
MUST KNOW THE CURSOR-ADDRESSING PREFIX AND OFFSET FOR YOUR CRT AS
WELL AS THE CODES TO CLEAR THE SCREEN. THIS PROGRAM IS DESIGNED
TO CHANGE BAZIC FOR THE TERMINALS LISTED AND TO ALLOW CUSTOM
CHANGES IF YOU KNOW THE CODES FOR A CRT THAT IS NOT LISTED HERE.
SOME CRTS MAY REQUIRE REWRITING THE CURSOR ADDRESSING AND CLEAR
SCREEN ROUTINES. THIS PROGRAM CAN BE CHANGED SO THAT IT 'WRITES'
THE PROPER ROUTINE BY ADDING AN ENTRY TO THE TABLE OF DATA
STATEMENTS AT THE END OF THIS PROGRAM. THE FIRST FIELD IS THE
NUMBER OF PAIRS THAT HAVE TO BE WRITTEN. EACH PAIR CONSISTS OF
THE VALUE TO BE WRITTEN AND THE NUMBER OF BYTES FROM THE
BEGINNING OF BAZIC OF WHERE THE BYTE IS TO BE WRITTEN.
```

THE BAZIC DISK MUST BE IN DRIVE ONE FOR THIS PROGRAM TO OPERATE.

PRESS RETURN TO CONTINUE

Press Return and the program will display a menu of 6 CRT options and an "other" option.

- 1=ZENITH Z-19 OR HEATH WH-19
- 2=ADM-3A
- 3=INTERTEC INTERTUBE
- 4=HAZELTINE
- 5=SOROC
- 6=NORTHSTAR ADVANTAGE
- 7=OTHER

If your CRT is not one of those listed consult your CRT manual for the values of your CRT and/or your programmer. We will, for the purposes of this example, select 5 for SOROC.

Next you will see:

- 1= 8 DIGIT SOFTWARE FLOATING POINT BAZIC
- 2= 10 DIGIT SOFTWARE FLOATING POINT BAZIC
- 3= 12 DIGIT SOFTWARE FLOATING POINT BAZIC
- 4= 14 DIGIT SOFTWARE FLOATING POINT BAZIC
- 5= 8 DIGIT HARDWARE FLOATING POINT BAZIC
- 6= 10 DIGIT HARDWARE FLOATING POINT BAZIC
- 7= 12 DIGIT HARDWARE FLOATING POINT BAZIC
- 8= 14 DIGIT HARDWARE FLOATING POINT BAZIC

ENTER NUMBER OF BAZIC TO BE CHANGED (0 TO END) 2

We will select 2 for 10-digit software floating point bazic. (Hardware floating point bazic is for use with the NorthStar Floating Point Board.)

Now you will see:

THIS LINE SHOULD BE AT THE TOP OF THE CRT AND THE REST OF THE SCREEN SHOULD BE CLEAR. 1=YES, 0=NO 1

Enter 1 if the above sentence is at the top of the screen and the rest of the screen is clear.

At this point a series of loop-de-loops will appear on the screen. Below the loop-de-loops the following will be displayed on the screen:

THIS COMPLETES THE TEST. IF EVERYTHING IS OK THERE SHOULD BE A DISTINCT PATTERN FROM THE PREVIOUS !@ STATEMENTS
ENTER 1= EVERYTHING IS OK, 0= NOT OK

If everything is okay enter 1 and the next prompt will be:

READY

Now enter the following (your responses are underlined):

```
READY
BYE <CR>
A>REN BAZIC10.COM=BAZIC10.001 <CR>
A>BAZIC10 <CR>
READY
```

Configuration is now complete. To test the clear screen function type (your response is underlined):

```
READY
CLS <CR>
```

The screen should clear and

```
READY
```

should appear at the top, left-hand corner of the screen.

Adapting baZic for operation under the MP/M I operating system:

The configuration process is the same as for CP/M.

For baZic to function under the MP/M I operating system, you must disable Control C. You can disable Control C by adding this line of code as the first line number in the first program to be run:

```
1 FILL 280,1
```

You must either add this line of code to the program or FILL this location manually for baZic to operate under the MP/M I operating system. To disable Control C manually (your response is underlined):

```
READY
FILL 280,1 <CR>
READY
```

Adapting baZic for operation under the MP/M II operating system:

Control C must be disabled (FILL 280,1, as in the MP/M I example above).

Through DDT, change the following locations in baZic from FF Hex to FD Hex:

```
0319 Hex
1C00 Hex
1F24 Hex
```

baZic File Name Extensions

Under CP/M and MP/M, baZic programs will have a ".002" extension, e.g., "PROGRAM.002." baZic data files generally will have a ".003" extension, unless specified otherwise. Type 1 machine language programs will have a ".001" extension.

INTRODUCTION

baZic was created by Micro Mike's, Inc. to provide a North Star-compatible BASIC interpreter which allows execution of North Star BASIC programs without modifications. Version II of baZic is designed to run under North Star DOS, CP/M^(R) and Micro Mike's, Inc. Disk Operating System (MicroDoZ). Some of the advances of baZic over BASIC include much faster execution of BASIC programs and increased power by the addition of new statements and functions.

All North Star features are supported through release 5.2 of North Star BASIC. However, baZic will not run programs written with BASIC 5.2 which use the FILESIZE or FILEPTR functions since baZic defines their associated operation codes (op codes) differently. Additional baZic features include: APPEND as an executable statement, ON GOSUB, additional print formatting capabilities, cursor addressing, clear screen, INSTAT and OUTSTAT, and a statement that allows all MicroDoZ commands to be executed from baZic.

All references to North Star BASIC will refer only to BASIC while references to Micro Mike's, Inc. Z80 BASIC will be by its trade name, baZic.

Although baZic has several new features, it is upward-compatible with BASIC. Upward-compatible means that all programs written under BASIC can be run under baZic with no modifications. The only exceptions to this rule are the few cases where programs use FILLS or CALLS to BASIC locations which will be different under baZic.

baZic was written to utilize the full Z80 instruction set. Because baZic is written entirely in Z80 code and changes have been made internally in the way baZic handles information, there are small external differences between BASIC and baZic that will affect programs and programmers.

Other than the increased performance (see Appendix 1), the only external changes result from the reassignment of operation codes for reserved words and the addition of new statements and functions. Specifically, the direct commands no longer have an operation code (op code) associated with them to allow the definition of more statements. This change has no effect upon the way programs operate but programmers will notice the change when reserved words are encountered in REMark statements.

In many cases garbage will be substituted for occurrences of reserved words in the REMark statements of programs written under BASIC and LISTed under baZic. This problem has no solution except to type the correct words back into the program using baZic instead of BASIC.

The control stack has been relocated under baZic to allow the use of Z80 block move instructions. The effects of this change are that the first reference to variables in a baZic program may take slightly more time than under BASIC, but all further operations involving these variables will be much faster under baZic than under BASIC.

The function look-up table has been expanded under baZic from 32 to 64 entries to allow the definition of more built in functions. This addition assures that baZic will not be soon obsolete and that many new functions can be added in the future as they are needed.

Because baZic has a new statement controlling cursor addressing, the "at" sign (@) is no longer used to cancel a line while editing a line of baZic code. Instead, the use of control N (^N), as in BASIC, causes the line to be cancelled.

This manual is NOT designed to teach BASIC but rather to describe the commands, statements, and function of baZic to those familiar with programming in North Star BASIC or other sophisticated BASICS. If you do not know BASIC, please use one of the many books on the market which teach BASIC before you try to use this manual. (Micro Mike's, Inc. currently has a beginner's manual under development.)

1.1 Definition of Mnemonics

To make the syntax examples of commands and statements easier to recognize, several mnemonics are defined in this section. For each command, statement, or function, the syntax will be given at the beginning of the description, followed by the specific meaning of each argument passed to the commands, statements, and functions.

All arguments passed will be surrounded by the "less than/greater than" (<>) symbol pair. Any argument with brackets ([]) surrounding the argument is an **optional argument** and need not be included in any use of the command or statement.

A list of the mnemonics and their meanings follows:

LINE#	baZic line number
DEVICE#	Number of an input or output device
DRIVE#	Number of a disk drive
FILENAME	The name of a file
#EXPR	A numeric expression
LOGEXPR	A logical expression
TYPEEXPR	A type expression
CHANNEL#	A disk channel number

1.2 Definition of Terms

This section is designed to minimize ambiguity by defining the numerous terms to be used throughout this documentation.

Arguments are values passed to commands, statements, and functions (both built-in and user defined). The arguments for commands, statements, and functions will be different but are generally similar in that the command, statement, or function will not operate correctly unless these values are given. Often arguments are optional and thus do not have to be passed. Arguments can easily be recognized in the syntax example of each command, statement, or function because they are enclosed by less than/greater than signs (<>).

Bits are units of storage used by the computer. A bit is a single yes/no response to an unambiguous question. Every operation a computer performs can be reduced to a series of true/false situations that represent a bit or combination of bits. Generally if a situation is true, the bit in question is a one and if the situation is false, the bit is zero.

Blocks are defined in this manual to be a unit for storing data on disk files that is 256 bytes long. This convention was initiated by North Star in the pre-double density days and has remained despite the introduction of 512 (and greater) byte sectors.

Bytes are units of storage used by the computer. A byte is eight bits long and is generally equivalent to a single character. Each character typed into a baZic program occupies one byte of storage both in the internal memory and on disk unless the character is part of a baZic reserved word (statement or function). In the case of a reserved word (such as PRINT), the entire word is converted to an operation code (op code) and is stored as a single byte.

Constants are data stored within a baZic program that are "constant" and never change unless the program is changed. Constants occur in DATA statements and can be used in all statements and functions to signify a value which will not change. Good programming techniques dictate the use of as few constants in a program as is possible.

The **current program** is defined as the program that is presently residing in the internal Random Access Memory of the computer. This program is different than the disk file programs (Type 2 files) in that the current program is lost if the computer's power is turned off. The internal (current) program is the one operated on by all direct commands. A program becomes the current program when it is LOADED from a disk or when the programmer types a line number followed by any program statements while creating a program.

Device Numbers are used to refer to an input or output device such as a printer or CRT. Eight devices are supported by baZic (0-7) except the CP/M version which supports four devices (0-3). A number sign (#) always precedes a device number in a baZic program. The device numbers are defined in the appropriate disk operating system (MicroDoZ or CP/M).

Drive Numbers are used to refer to disk drives. As many as seven drives are supported (1-7). If your system has a hard disk drive and uses Micro Mike's, Inc. JOEMASTER, your hard disk can be defined as virtually any number of "drives." However, only seven drives can be "looked at" by baZic at any single moment.

File Channels are used by baZic when files are OPENED by a program to allow the file to be referred to by a number only. Eight channels can be defined (0-7). An internal buffer is established for each channel opened.

File Names can be any combination of letters, numbers, or special symbols that uniquely define (name) a file. The file name must be eight characters or less and must contain no commas or spaces. File names for CP/M contain an optional decimal point followed by a three character type extension (".003", ".002", etc.).

File Type numbers are used to distinguish between different kinds of files. File types may range from 0 to 127. The only types normally defined are: 1 for assembly language, 2 for a baZic program, and 3 for a baZic data file. Under CP/M, these conventions are retained as close as possible by defining the type extension such that: .001 is for assembly language, .002 is for a baZic program and .003 is for a baZic data file.

I/O refers to the process of Inputting or Outputting information. The CRTs and printers are the main devices that cause or accept I/O but many other devices can perform these functions.

Legal will be used many times in different contexts. The overall meaning of "legal" is "by the established rules." Legal may be used in reference to line numbers (must be in the range of 0 to 65535, etc.), filenames (must have eight letters or less), drive number specifications (drives 1-7), etcetera.

Line Numbers are used in a baZic program to indicate the order of processing. baZic programs are always executed from the lowest to the highest line number unless a branching statement is encountered. Line numbers must be positive integer values in the range of 0 to 65535. Program statements or functions are assigned to a specific line number and multiple statements are allowed for each line number if they are separated by a backslash (\).

Numeric Expressions are any combination of numeric constants, numeric variables, array variables, subscripted array variables, or numeric functions, inclosed in parentheses, joined together by one or more arithmetic, logical, or relational operators in such a way that the expression, as a whole, can be reduced to a single numeric constant when evaluated.

Sectors can be any value and are hardware dependent. Double density and quad capacity North Star disk drives have 512 bytes per sector while single density North Star disk drives have 256 bytes per sector. All hard disk units supported by Micro Mike's, Inc. are 512 bytes per sector. The user of an application package should not normally need to know the sector size of the equipment.

Variables are combinations of letters and numbers representing data within a program. Legal variable names are the letters of the alphabet alternately followed by a number from 0 to 9. String variable names are similar except a dollar sign (\$) is placed after the letter and number to show the variable is a string variable. Variables can and do change their values many times during the execution of a program.

DIRECT COMMANDS

Direct Commands may be executed only from the direct mode. In the direct mode, no line numbers are used and commands are executed immediately. Several Statements also can be executed from the direct mode. These will be discussed in Section 3 (STATEMENTS).

The following Statements may be executed in the direct mode as direct commands:

APPEND	DIM	LINE
CHAIN	CREATE	DESTROY
IF THEN ELSE	PRINT	PRINT@
RESTORE	OUT	OPEN
CLOSE	READ#	WRITE#
CLS	FILL	LET
DOSCMD (MicroDoZ version only)		

2.1 Programming Commands

Programming Commands are commands used in the act of writing a baZic program. These commands generally are actions that "do something" to a baZic program such as LIST it, DElete line numbers, RENumber line numbers, AUTOMatic generation of line numbers, SCRatch a program, Program SIZE, or SET MEMory for larger programs.

2.1.1 LIST a program

```
LIST [#<DEVICE#>] [,<LINE#>] [,<LINE#>]
```

The purpose of the LIST command is to output the listing of the current program to an output device (usually the CRT or printer). The DEVICE# is a legal output device (0-7) preceded by the number sign (#). If the DEVICE# is not used, the default device is printed to (device 0).

The LINE# is the program line number to start listing from or a range of line numbers that you specifically want listed. If only one line number is specified, baZic will list only that line. If only one line number is specified, **followed by a comma**, all line numbers from the specified line number to the end of the program will be listed.

If two line numbers are specified (separated by a comma), baZic will list the range of line numbers included. If a specified line number does not exist, baZic will find the next larger line number and execute the LIST command as if the larger line number had been entered by the user.

If the optional line numbers are not specified, baZic will LIST the entire program currently in memory.

Examples of the use of the LIST command are:

LIST	(LIST all line numbers)
LIST 100	(LIST line number 100 only)
LIST 100,	(LIST line number 100 to end)
LIST #2	(LIST all line numbers to device #2)
LIST #2,100,500	(LIST lines 100 to 500 on device #2)

2.1.2 DElete line numbers

DEL <LINE#>,<LINE#>

This command is used to DElete all program lines (from the current program) that fall between the two line numbers, including the two line numbers passed as arguments to this command. The two line numbers must be legal line numbers in the current program.

The second line number must always be greater than the first or an ARG ERROR will be returned. If one or both of the specified line numbers do not exist, a LINE NUMBER ERROR will be generated. The value of line numbers must always be in the range of 0 to 65535 or an OUT OF BOUNDS ERROR will occur.

If the programmer wants to DElete only one line number, type only the line number to be deleted followed by a carriage return.

All variables within the current program are cleared upon the completion of a DElete command.

Examples of the use of the DElete command are:

DEL 10,100	(DElete lines 10 through 100)
DEL 1,2	(DElete lines 1 and 2)

2.1.3 SCRatch a program

SCR

The SCRatch command is issued to cause the current program and all its associated variables to be SCRatched from internal memory. This program cannot be recovered from RAM so if you want to use it again make sure the program has been previously SAVED on disk. The syntax guide serves as an example since no arguments are passed to this command.

2.1.4 RENumber a program

```
REN [<LINE#>] [,<INCREMENTAL VALUE>]
```

The RENumber command causes the line numbers of a program to be RENumbered. All program references to a line number within the program (GOTO, GOSUB, RESTORE, etc.) will also be RENumbered so that the program will continue to execute properly after the program has been RENumbered. If there are references to line numbers not present within the current program, these references will **not** be changed.

The LINE# argument must be a legal line number in the current program from which RENumbering is to start and the INCREMENTAL VALUE must be the optional value you want between line numbers.

If no arguments are passed to this command, baZic will RENumber starting with the line number 10 and automatically incrementing by 10. The LINE# argument must be a positive integer in the range of 0 to 65535 and the INCREMENTAL VALUE must be a positive integer. No line number in the RENumbered program may be greater than the largest legal line number (65535). If any error conditions occur while trying to RENumber a program, the program will NOT be RENumbered.

Examples of this command are:

```
REN                (Start with 10, increment by 10)
REN 100,20         (Start with 100, increment by 20)
REN 15             (Assumes INCREMENTAL VALUE of 10)
```

2.1.5 AUTO line numbering

```
AUTO [<LINE#>] [,<INCREMENTAL VALUE>]
```

This command causes baZic to AUTOMATICALLY generate line numbers as the programmer is typing in program statements. The line numbering will start with the LINE# argument and will increment each succeeding line number by the specified amount. If neither value is specified, baZic will start at line 10 and increment by 10. If no incremental value is specified, baZic will start at the number specified and increment by 10.

The LINE# argument must be a legal line number from which you want AUTOMATIC line numbering to start. The INCREMENTAL value is the amount by which you want each succeeding line number generated to be incremented. The LINE# and the INCREMENTAL VALUE must both be positive integers in the range of 0 to 65535.

To terminate AUTO line numbering, press the return key immediately after the line number before any other key is pressed. If any lines in the current program have the same line number as those generated by the AUTO command, the old line will be replaced by the new line.

Examples of AUTOMatic line numbering are:

AUTO	(Start at line 10 and increment by 10)
AUTO 200	(Start at line 200 and increment by 10)
AUTO 300,15	(Start at line 300 and increment by 15)

2.1.6 Program SIZE

PSIZE

This command causes baZic to return the size in blocks (256 bytes) of the current program in memory. This command can be used to determine if the current program will "fit" in a file previously CREATED. The command syntax guide serves as an example since no arguments are passed to this command.

2.1.7 SET MEMORY upper limits

MEMSET <MEMORY ADDRESS>

MEMSET is used to adjust the high limit of internal memory. The MEMORY ADDRESS must be the Decimal address of the new memory high limit. The current program is left unchanged but all variables will be lost. The baZic in memory is modified so that a disk copy can be obtained by saving baZic from its operating location in RAM to a properly created disk file. See your disk operating system manual if you are uncertain how to save a file from the operating system.

Examples of the MEMSET command are:

MEMSET 24575	(5FFF Hex)
MEMSET 32767	(7FFF Hex)
MEMSET 40959	(9FFF Hex)
MEMSET 49151	(BFFF Hex)
MEMSET 57343	(DFFF Hex)

2.2 Disk Commands

The following commands (with the exception of BYE) affect the disk drives. If the command BYE is executed from direct mode, baZic will return to the Disk Operating System where all disk commands are available. The other commands cause the CATalog of the specified disk to be displayed or programs to be LOAded from the disk to internal memory or programs in internal memory to be SAVEd on the disk.

2.2.1 CATalog a disk

CAT [#<DEVICE#>] [<DRIVE#>] [,WILDCARD]

The CATalog command is used to "view" the directory (catalog) of the disk drives. The device number must be a legal output device (0-7) and the drive number a legal drive for your system hardware (1-7). Both values must be a positive integer in the proper range.

The DEVICE# must be preceded by a number sign (#). If no device is specified the default device is used (device #0). If no drive number is specified, baZic will assume the default drive (Drive #1).

The WILDCARD parameter may be passed, in the MicroDoZ version only, to view only those files which match the wildcard sequence. If the character "T" is used, only those files beginning with the character "T" will be displayed. More than one character can be used as the wildcard. If the user wants to match a character occurring in the middle of a file name, an asterisk (*) may be used to precede the character to be matched.

Examples of the CATalog command are:

```

CAT                (CAT default drive to default device)
CAT #2             (CAT default drive to device #2)
CAT 5              (CAT Drive 5 to default device)
CAT #2 5           (CAT Drive 5 to device #2)
CAT #2,5 MicroDoZ only (CAT Drive 1 to device #2 only those
                    files beginning with 5)
CAT 1,*I MicroDoZ only (CAT only those files with I as second
                    character)
    
```

The CATalog of a disk will appear different under each operating system that baZic can run under. The CAT command under MicroDoZ results in the following information being given for each file listed in the directory of the affected disk:

```

File Name (maximum of eight characters)
Starting Disk Address (Decimal)
Length of the File in Blocks (Decimal)
Density (Single or Double)
Type of File (0 to 127 Decimal)
GO Address (If Type 1 in Hex)
R/O if Read Only File
SYSTEM if System File
Attribute Field (0 to 63 Decimal)
    
```

A sample CATalog under MicroDoZ would appear as follows:

READY
CAT

MICRODOZ	4	20 D	0		AF	0
M2D00M	14	10 D	1 2D00		AF	0
TEST	19	2 D	2	R/O SYSTEM	AF	0
BAZIC	20	54 D	1 0100		AF	0
READY						

A CAtalog of the disk can be obtained from a program by use of the DOSCMD statement (MicroDoZ version only). The statement would appear as follows:

10 DOSCMD "LI"

Under CP/M, the only items listed following a CAT command are the file name and its extension. A CAtalog under CP/M will appear as follows:

READY
CAT

: BAZIC08F COM : BAZIC10F COM : BAZIC12F COM : BAZIC14F COM
: CRT 003

2.2.2 SAVE a program

SAVE <FILENAME>

The SAVE command is used to store the current program in a type 2 disk file. The FILENAME must evaluate to a legal file name. If the current program is larger than the specified file, an OUT OF BOUNDS error will be generated. The SAVE command does not change the current program as it is being saved.

If the current program is a new program and has no file created for it, the command NSAVE must be used instead of SAVE. The size of the current program can be determined by using the PSIZE command and the size of the program file can be determined by the CAT command.

Examples of the SAVE command are:

SAVE PROGRAM
SAVE CONTROL,5

2.2.3 SAVE a New program

NSAVE <FILENAME> [<FILESIZE>]

The NSAVE command is used when the current program is a new program or an old program is too large to fit in its existing file. NSAVE causes a new type 2 directory entry to be made with the specified name. If the optional FILESIZE argument is not passed, the file is created 3 or 4 blocks longer than the Program SIZE of the current program. (Double density files must always be an even number of blocks.) The FILESIZE argument can be used to create the file of any legal file size.

Under CP/M, NSAVE is still used but the context is slightly different. Normally, NSAVE would not be needed under CP/M because CP/M is dynamic and the file "grows" as the program expands. To retain as much compatibility with the MicroDoZ version, NSAVE is still a valid command. If the file already exists and an NSAVE command is issued, baZic will return an ARG error to inform the programmer of his/her mistake.

Examples of the NSAVE command are:

```
NSAVE CONTROL      (Save on default drive, PSIZE+3)
NSAVE CONTROL,5   (Save on Drive 5 to file PSIZE+3)
NSAVE MENU 20     (Create file 20 blocks long)
NSAVE MENU,2 30   (Save on Drive 2 file 30 blocks long)
```

2.2.4 LOAD a program

LOAD <FILENAME>

The LOAD command is used to LOAD the specified file from the disk into internal memory. The program now becomes the current program. Any previous "current program" is SCRatched and all variables are cleared upon a LOAD command. The specified file must be a type 2 file.

If the specified program is too large to fit into internal memory or there is no valid program in the file (a valid end of file marker is not found), a TOO LARGE OR NO PROGRAM ERROR is generated.

Examples of the LOAD command are:

```
LOAD MENU,2       (Load "MENU" from Drive 2)
LOAD CONTROL
```

2.2.5 BYE to disk operating system

BYE

The BYE command is used to exit from baZic to the disk operating system. Upon the execution of the BYE command the disk operating system prompt will be displayed. See the disk operating system manual for more information.

When baZic is exited using BYE, the current program (if any) is not disturbed and may be re-entered if it is not changed by the execution of operating system commands. baZic has three entry points. The entry points are given in relation to the Origin (ORG) of the baZic. The normal origin for baZic under MicroDoZ and CP/M is 0100 Hex.

The following table describes the three entry points for baZic.

Hex Address	Entry characteristics
ORG	Clears program and all variables
ORG + 04 Hex	Retains the program but clears the variables
ORG + 14 Hex	Retains the program and variables

The command syntax serves as the example since no arguments are passed to this command.

2.3 Execution Commands

The following two commands cause baZic to begin executing a program. The RUN command is generally used to initiate a program execution but the CONTInue command can be used to re-start a program that was stopped by the STOP statement or a control C. The APPEND and CHAIN statements also can affect the execution of programs.

2.3.1 RUN a program

RUN [<LINE#>]

The RUN command is used to execute the current baZic program. If the optional LINE# argument is not passed, baZic "looks" for the first line number to begin execution. If the LINE# argument is passed, execution begins on the specified line.

The execution of the RUN command causes all variables to be set to their default conditions.

Examples of the use of the RUN command are:

```
RUN
RUN 500
```

2.3.2 CONTInue a program

CONT

The CONTInue command is used to resume execution of a program that was stopped by the STOP statement or a control C by the user. If the statement where program execution was stopped is an INPUT statement, CONTInue causes the program to begin executing the INPUT statement again. Otherwise CONTInue causes the next statement after the interruption to begin executing.

The current program cannot be changed after a program is stopped and before the CONTInue command is executed, but variables in a program can be changed before the CONTInue command is invoked. The program cannot be CONTInued after a program error or an END statement is encountered.

The syntax line serves as an example since the user must simply type the letters "CONT" for a program to continue.

2.3.3 Execution Statements

Two statements can be used to cause programs to be executed from other programs. These statements are mentioned in this section because they are very similar in their actions to the RUN command. The CHAIN statement can be used as a command which has the same effect as issuing the two commands LOAD a program and RUN it. In addition, the CHAIN command can be used to transfer control from one program to another by CHAINing.

The APPEND statement also can be used to change control from one program to another. If the user has a series of subroutines or functions common to more than one program, he/she can position these routines in the beginning line numbers and APPEND the remaining program. Programs can be changed by using the APPEND statement with the optional line number command to cause only that portion of the specified program to be exchanged, leaving the common portion in memory.

See the statements CHAIN (Section 3.4.8) and APPEND (Section 3.4.7) for more information.

STATEMENTS

Statements are the "stuff" programs are made of. To define a program to do any useful task, the task must be divided into many exact operations. A statement is an exact operation. Statements direct the flow of the program and cause variables to be exchanged between the user and internal memory and between internal memory and external disk storage. A series of statements placed sequentially in a logical manner is a program.

Statements begin with the statement name and are optionally followed by arguments. Many arguments are optional and some statements take no arguments at all. If arguments are included, they change the way the statement executes.

3.1 Program Data Statements

The following three statements are concerned with the storage of data within a program. Constant data can be stored in DATA statements within a program. An internal pointer can be set to "point" to different sets of data with the RESTORE statement and the data can be passed to variables within the program by the READ statement.

3.1.1 DATA constants

DATA <LIST OF CONSTANTS>

The DATA statement is used to define constant data within a baZic program. The LIST OF CONSTANTS can be numeric or string data in any combination with each data element separated by commas. All data stored as strings must be enclosed in quotation marks.

DATA statements may be READ by a program to pass the constant information to program variables. DATA statements may be placed anywhere within a program and are "passed over" by baZic if encountered while executing a program. If DATA statements will be used only a few times in a program, place the DATA statements near the end of the program to speed the execution of the program.

If a READ statement cannot READ a DATA statement, a SYNTAX or READ error is returned depending upon the problem encountered.

If a DATA statement is encountered during program execution, it is "ignored." The only effect upon the program is to slow execution. If a DATA item or items are to be READ many times in a program, it may be advantageous to locate the DATA statements close to the beginning of the program.

Examples of the use of the DATA statement are:

```
DATA -1,"ENTER OPTION NUMBER",.42,0,2,0,99,1
DATA "FIRST MESSAGE","SECOND MESSAGE"
```

3.1.2 READ data constants

```
READ <LIST OF VARIABLES>
```

The READ statement causes the specified variables to be "filled" by the constant data listed in the DATA statement. The READ is always sequential in that the first DATA element is read followed by the next until the end. The only variation to this is when a RESTORE is executed between READS.

READS must always match data types. A numeric variable must always READ a numeric constant and a string variable must always READ a string constant.

The DATA elements are always pointed to by an internal pointer in baZic. When the program is first RUN the pointer will always "point" to the first DATA statement in the program. This pointer can be changed by using the RESTORE statement.

As each DATA element is READ into a variable, the pointer is advanced automatically to the next DATA item. The DATA pointer doesn't care if the next DATA element is on the same line as the previous element or separated by many lines and always "points" to the next logical DATA element.

The DATA pointer "knows" when the last DATA element has been READ. If an attempt is made to READ past the end of data, baZic will find the end of program mark before finding the additional data. Attempts to READ beyond the end of data will generate a READ ERROR. If the DATA elements are to be READ again, a RESTORE must be executed.

A short program is given as an example of the READ statement:

```
10 FOR N=1 TO 3
20 READ X
30 PRINT X
40 NEXT N
50 RESTORE 100
60 READ X$
70 PRINT X$
80 DATA 1,2,3
90 DATA 4,5,6
100 DATA "THE END"
```

When this program is RUN the output would appear as follows:

```

READY
RUN
1
2
3
THE END
READY

```

3.1.3 RESTORE data pointer

```
RESTORE [<LINE#>]
```

The RESTORE command is used to change the data pointer so that it "points" to the specified DATA statements. If no line number is specified, the first (lowest number) line number containing DATA statements is pointed to. (The pointer is not actually set until the first READ statement is executed.) If a line number is specified, the pointer "points" to the first DATA element in that line number.

The example in Section 3.1.2 (READ data constants) shows the use of the RESTORE statement.

3.2 Input and Output Statements

Input and output statements cause information to pass between devices such as CRTs or printers and internal memory. In some systems, file information is "input." However, baZic has separate file statements which cause information to pass between files and internal memory.

3.2.1 PRINT a variable

```

PRINT [#<DEVICE#>] [,<LIST OF EXPRESSIONS>]
or !      [#<DEVICE#>] [,<LIST OF EXPRESSIONS>]

```

The PRINT (or ! for shorthand print) statement causes string or numeric variables to be output to a device (normally a printer or CRT). If no device expression is used, the default device is assumed. If the device expression is used, it must be preceded by the number sign (#) and followed by a comma. The device expression must evaluate to a positive integer from 1 to 7.

If no LIST OF EXPRESSIONS is specified, a carriage return and line feed is all that is output. If the LIST OF EXPRESSIONS is followed by a comma, the carriage return is suppressed and additional PRINT statements will continue printing on the same line. Each element in the LIST OF EXPRESSIONS must be separated by a comma.

Other versions of BASIC use a semicolon (;) to signify no carriage return after printing a series of variables, but in baZic the comma (,) has this function. The comma in these other versions of BASIC is used to TABulate to preset tab stops but in baZic the TAB function can accomplish the same job.

A sample program and RUN follow as an example of the PRINT statement:

```

5  D=1                                \REM 1 IS THE PRINT DEVICE
10 PRINT "THIS IS A TEST"
20 FOR N=1 TO 3
30 PRINT N,                            \REM SUPPRESS THE RETURN
40 NEXT N
50 PRINT                                \REM SINGLE PRINT TO ISSUE RETURN
60 PRINT #D,"TO PRINTER" \REM PRINT TO PRINTER
70 !"THE END"                          \REM USE ! INSTEAD OF PRINT

```

When this program is RUN the CRT (device #0) would appear like this:

```

READY
RUN
THIS IS A TEST
1 2 3
THE END

```

and the printer (device #1) would appear like this:

```

TO PRINTER

```

3.2.1.1 Formatted Printing

The formatted printing capabilities of baZic are used when default printing of numeric information is not desirable. Default printing involves printing a space character () and printing of a number without regard to its characteristics.

As an example, if we want to print a dollar amount that was calculated to be 45.586889, we would normally not want the extra digits to be printed. Also, we would want the number to be preceded by a dollar sign (\$) and if printed in a column with other dollar amounts, printed so that the decimal points "line up." In this instance, we would want the number printed using the format feature of baZic.

baZic can switch automatically to formatted output in certain instances. This switch occurs when the number to be printed is too large or too small to be printed by the current precision of baZic. baZic will print these numbers in the E (or exponential) format automatically.

Formatted printing involves the inclusion of **format characters** within the print statement to "tell" baZic how you want the numbers to be printed. If no format characters are specified the variable will be printed in "free format."

In "free format", baZic is free to choose which format (normal or exponential) is best and the number of characters printed will vary according to the number of characters in the number. The field length is always set to 1 greater than required to print the number to allow for the assumed plus sign (+). The number to its full precision is printed right justified with all trailing zeros suppressed.

Under formatted printing, the print field is expressly defined and each subsequent print under the same format will result in the same number of characters being printed. Three formats can be specified: F format, I format, and E format.

The F format (nFm) is used to print numerics which have a Fixed decimal place. The field length is specified as "n" and the number of digits to the right of the decimal is "m." This format will always be printed right justified with the printing starting "n" digits to the left of the right margin.

The I format (nI) is the Integer format. The field will be "n" characters wide, but will have no decimal point. This format will always be right justified. Only integer values can be printed under this format. Attempts to print decimal values will result in a FORMAT ERROR.

The E format (nEm) is the Exponential format. The field will be "n" characters wide and will be right justified; "m" digits will be to the right of the mantissa decimal point.

When using print formatting, the programmer tells baZic that formatted printing follows by the use of the percent sign (%). When "%" is used in a PRINT statement, the output is to be formatted. The syntax of print formatting is as follows:

```
PRINT [<%FORMAT CHARACTERS FORMAT SPECIFICATION>],<VARIABLES>
```

Several characters are defined in baZic to have special meaning when they appear in the format syntax. These characters are called **FORMAT CHARACTERS**. Format characters can be mixed in a formatted print. They must always come after the percent sign (%) and before the **FORMAT SPECIFICATION**. A list of **FORMAT CHARACTERS** follows:

- A A is the Accounting format. All negative numbers are printed with a "less than/greater than" pair (<>) around the number.
- C C is the Comma format. Commas will be inserted after each group of three digits to the left of the decimal in numbers large enough to warrant this attention. This format character does not have an effect in the E-format.

- Z Z is the suppress Zeros format character. All zeros trailing the decimal point will be suppressed and spaces will be printed instead.
- + + is the positive number format character. All positive numbers are printed with the plus sign (+) to the left of the number.
- \$ \$ is the dollar sign format character. This format character causes a dollar sign (\$) to be printed to the left of the number.
- # # is the default format character. The appearance of the # format character in a format field will cause the current format to become the default format. This means that all further PRINTs will print in the specified format even though no further format specifications follow. A %# format causes free format to be reinstated.

The print field is calculated to be all the digits, the decimal point and any other characters specified such as a dollar sign or comma. Care must be used in calculating the field length as any number which generates more digits (including the format characters) than the specified field length will result in a FORMAT ERROR.

Examples of print formatting follow:

FORMAT	VALUE	RESULT UPON PRINTING
%A8F2	19.355	19.36
%A8F2	-19.355	<19.36>
\$\$CA13F2	201758.88	\$201,758.88
\$\$CA13F2	-201758.88	\$<201,758.88>
%C5I	1000	1,000

3.2.1.2 PRINT@ and !@

PRINT@ (ROW,COLUMN)
 or !@ (ROW,COLUMN)

PRINT@ and !@ are further enhancements of the print statements of baZic. This print statement is CRT-specific and will work only if baZic has been "set up" for your CRT. The CRT program is used to establish the proper cursor addressing prefix for the CP/M version of baZic. For MicroDoZbaZic the program CONFIG serves the same purpose.

To use this print function, pass the ROW and COLUMN coordinates of the position on the CRT where you want the cursor to be positioned. If your CRT does not support cursor addressing, do not use this print feature. The ROW and COLUMN values can be numeric expressions as long as they evaluate to a legal ROW and COLUMN number.

Examples of the INPUT statement follow:

```
10 INPUT A
20 INPUT "ENTER OPTION (1 TO 10) ",X
30 INPUT #2,X$
40 INPUT "WHAT IS YOUR ADDRESS ",A$
```

3.2.3 INPUT1 a variable

```
INPUT1 [#<DEVICE#>] [,<STRING PROMPT>,<VARIABLE LIST>
```

The INPUT1 statement is identical to the INPUT statement except the INPUT1 input does not echo a carriage return and line feed when the user makes an entry. This input is used when multiple inputs are required on the same line. The examples for INPUT1 are the same as INPUT.

3.2.4 OUTput a byte

```
OUT <PORT NUMBER>,<BYTE VALUE>
```

The OUT statement is used when the programmer wants to output information directly to a specific Z80 port. The PORT NUMBER and the BYTE VALUE must be constants or numeric expressions which evaluate in the range of 0 to 255 (the range of a byte). This statement (as do all baZic statements) uses decimal numbers.

3.3 Branching Statements

In baZic, the "normal" flow of processing is from the smallest line number to the largest line number. This situation is rarely effecient in writing a program. Some method must be available to allow the program to branch from one section to another. The following sections explain the branching statements of baZic.

3.3.1 GOTO a line number

```
GOTO <LINE#>
```

Upon execution of the GOTO statement, program flow is immediately branched to the specified line number. "Normal" execution resumes at that point unless another branching statement is encountered. The LINE# argument must be a positive integer in the range of 0 to 65535.

Since GOTO is a reserved word, GO and TO may not be separated by spaces as in some BASICS.

An example program is provided to demonstrate the GOTO statement. Notice that this program "branches forever" or until a control C is detected (if Control C is enabled).

```
10 PRINT "THIS IS THE FIRST MESSAGE"
20 PRINT "THIS MESSAGE WILL APPEAR MANY TIMES"
30 GOTO 20
```

3.3.2 GOSUBroutine

GOSUB <LINE#>

The GOSUB statement is one of the most powerful statements in baZic. A subroutine is a series of statements which need to be used over and over in a program. Therefore, these program lines are made into a subroutine. The difference between a GOTO statement and a GOSUB statement is a GOSUB always RETURNS to the statement immediately following the GOSUB statement, assuming that the subroutine has a RETURN instruction somewhere within the subroutine and the RETURN statement is executed.

The GOSUB statement can be used within a subroutine to call another subroutine. These GOSUB statements can be "nested" as deep as available memory allows. All good programmers make extensive use of subroutines because subroutines can help structure the program and conserve memory.

An example of the use of a subroutine follows:

```

10 !"THIS IS THE BEGINNING"
20 GOSUB 50
30 !"THIS IS THE END"
40 END
50 !"THIS IS THE SUBROUTINE"
60 RETURN

```

The results of RUNning this program would appear as follows:

```

READY
RUN
THIS IS THE BEGINNING
THIS IS THE SUBROUTINE
THIS IS THE END
READY

```

3.3.3 RETURN from a subroutine

RETURN

The RETURN statement is used to exit a subroutine and RETURN to the statement which follows the GOSUB statement that called the subroutine. No arguments are passed to this statement because baZic "remembers" the exact GOSUB statement which called the routine. An example is provided in the previous section under GOSUB.

A special form of the RETURN statement is used to return from a User Defined Function. See Section 6.2 for more details. A RETURN also "EXITS" all FOR NEXT loops within the subroutine. See Section 3.3.7 for more information.

3.3.4 ON value GOTO a line number

ON <#EXPR> GOTO <LIST OF LINE NUMBERS>

The ON GOTO statement is used when a program needs to branch to many line numbers from one line. The numeric expression (#EXPR) is evaluated to determine the line number in the list of line numbers the program is to branch. The #EXPR must evaluate to a positive integer from 1 to the maximum number of line numbers in the list. The maximum number is determined by how many line numbers can be placed on a line.

If the #EXPR evaluates to 1, the program branches to the first line number in the list. If the #EXPR evaluates to 2, the program branches to the second line number in the list, etc.

Examples of the ON GOTO statement follow:

```
50 ON X GOTO 100, 200, 300, 400 (X must be 1 to 4)
100 ON INT(Y+1) GOTO 10,20,30,40 (Y must be 0 to 3)
```

3.3.5 ON value GOSUBroutine

ON <#EXPR> GOSUB <LIST OF LINE NUMBERS>

The ON GOSUB statement is very similar to the ON GOTO statement except in this case a subroutine is branched to. Upon the occurrence of a RETURN instruction, control passes to the line number immediately following the ON GOSUB statement. An example program is:

```
10 FOR N=1 TO 3
20 ON N GOSUB 100,200,300
30 NEXT N
40 END
100 !"THIS IS SUBROUTINE ONE"\RETURN
200 !"THIS IS SUBROUTINE TWO"\RETURN
300 !"THIS IS SUBROUTINE THREE"\RETURN
```

3.3.6 IF THEN ELSE

IF <LOGEXPR> THEN <STATEMENT> [ELSE <STATEMENT>]

The IF THEN ELSE statement is used to branch based upon the results of the evaluation of a logical expression (LOGEXPR). (A Boolean Variable can also be used as a logical expression.) If the expression evaluates as true the THEN statement is executed. If the expression evaluates as false the ELSE statement is executed (if present) or the program flow executes the next sequential statement. See Section 8.3.2 (IF THEN Evaluation) for more information.

IF THEN ELSE statements can be nested.

Examples of the IF THEN ELSE statement are:

```

10 IF X=5 THEN 100
20 IF ABS(Y)>X THEN 210 ELSE 440
30 IF A$="YES" THEN PRINT "YES" ELSE PRINT "NO"

10 INPUT A
20 IF A THEN ! "YES" ELSE ! "NO"

```

If $A=0$, the preceding program prints "NO." If $A <> 0$, the preceding program prints "YES." The Boolean evaluation in this program would execute faster than the "normal" method of programming which would appear as follows:

```

10 INPUT A
20 IF A <> 0 THEN ! "YES" ELSE ! "NO"

```

3.3.7 FOR NEXT STEP EXIT

```
FOR <CONTROL>=<INITIAL> TO <LIMIT> [STEP <VALUE>]
```

The FOR NEXT loop is used to control programming situations where a similar process is taking place many times and is controlled by the CONTROL variable.

The CONTROL variable is used to determine the status of the loop. Each time through the NEXT statement, the value of the CONTROL variable is incremented (if no STEP value is given) by 1 and a comparison is made with the LIMIT value to determine if program flow should remain within the FOR NEXT loop or "fall through" to the following statement.

If the comparison indicates the CONTROL variable has not yet exceeded the LIMIT, the loop is executed again and the process repeated until the LIMIT is exceeded at which time program flow continues with the statement following the loop.

The STEP value is optional and can be used to cause the loop to be incremented by any value or even decremented (if STEP equals a negative number and the CONTROL variable is greater than the LIMIT). The STEP value adds extra flexibility to the FOR NEXT loop.

FOR NEXT loops may be "nested" to any level that memory allows. A FOR NEXT loop is nested when a second loop starts and ends entirely within a previous loop. The innermost loop must always be completed before trying to terminate any outside loops. The example will demonstrate nested loops.

FOR NEXT loops may be executed 0 times if the CONTROL variable already exceeds the LIMIT value when control passes to the loop. This feature can be very handy in programming, allowing a loop to be used or not depending upon the conditions set. If a loop is executed 0 times, it is not executed at all.

When using the NEXT statement at the end of a FOR NEXT loop, the argument variable can be omitted from the NEXT statement. This condition results in faster execution of the loop but offers more opportunities for the programmer to make mistakes. If the control variable is named in the NEXT statement, baZic makes a comparison with the CONTROL variable to determine if you are "NEXTing" the correct variable and associated loop.

If the programmer wants to leave a FOR NEXT loop without the CONTROL variable reaching the LIMIT value, an EXIT statement must be used. The EXIT has the effect of cancelling the "unused" loop and transferring program control to another line number. EXIT can be thought of as a GOTO out of FOR NEXT loop. If loops are nested, each loop must have an associated EXIT statement if that loop is to be EXITed prematurely.

The only exception to this rule is when a RETURN is executed from a subroutine or a user-defined Function. A RETURN causes all FOR NEXT loops within the subroutine or user defined function to be closed (EXITed).

The following example program demonstrates FOR NEXT loops, STEP values, the EXIT statement, nested loops, and the fact that the control variable is incremented the last time through the loop:

```

10 FOR N=1 TO 2 STEP .2
20   FOR M=1 TO 10
30     IF M=5 THEN EXIT 50
40     NEXT M
50 PRINT "N=",N,TAB(10),"M=",M
60 NEXT N
70 PRINT "N=",N

```

If the preceding program was RUN, the following results would appear on the output device:

```

READY
RUN

N= 1      M= 5
N= 1.2    M= 5
N= 1.4    M= 5
N= 1.6    M= 5
N= 1.8    M= 5
N= 2      M= 5
N= 2.2
READY

```

3.4 File Statements

The following statements allow the manipulation of disk data files. Statements are provided to CREATE, DESTROY, OPEN, CLOSE, READ, and WRITE data files. baZic can "look" at all file types as a data file, even if the data within the file is a machine language or baZic program. Files can be accessed in sequential or random fashion and information read or written as bytes, strings or numbers.

If you are using the CP/M version of baZic, all files which are to be accessed should have a numeric extension which is equivalent to the North Star convention. All baZic program files must have a ".002" extension and all data files must have a ".003" extension (unless the program specifies another numeric type).

The North Star convention of drive naming has been retained even in the CP/M version of baZic. Drive 1 is equivalent to CP/M Drive A, Drive 2 is equivalent to Drive B, etc. If the drive number is not specified, the default drive is assumed.

3.4.1 CREATE a file

```
CREATE <FILENAME>,<FILESIZE>[,<FILETYPE>]
```

The CREATE statement is used to create a file on the disk. A file is created with the specified FILENAME and FILESIZE. If the optional FILETYPE argument is passed, the file is created with that type. The FILENAME must be a legal file name.

The FILESIZE argument refers to file blocks and must be within the range of available space on your disk drives. This value can be as much as 16 megabytes (16,776,960 bytes) on a hard disk system but is hardware dependent.

The FILETYPE argument can be any positive integer number in the range of 0 to 127. If the FILETYPE is not specified, baZic assumes a type 3 file which is a "normal" baZic data file.

When the CREATE command is executed, a directory entry is made according to the specifications of the command call but no access is made to the actual file space.

Examples of the CREATE statement are:

```
CREATE "DATAFILE,2",100
10 CREATE "MENU1",20,2      (Create file for baZic program)
20 CREATE N$+D$,N1,N2
```

3.4.2 DESTROY a file

```
DESTROY <FILENAME>
```

The DESTROY statement is used to delete a file name from the directory of the specified drive. This command is equivalent to the DE command of MicroDoZ. The only action taken is the removal of the specified file name from the directory of its disk. The actual file is not changed.

In order to be compatible with non-CP/M versions of baZic and with BASIC, the DESTROY statement destroys all files of the specified name on the specified disk. If the specified file does not exist, an ARGument error is returned.

Examples of the DESTROY statement are:

```
DESTROY "DATAFILE,2"
10 DESTROY "MENU"
20 DESTROY N$+D$
```

3.4.3 OPEN a channel

```
OPEN #<CHANNEL#>[%<TYPEEXPR>],<FILENAME>[,<SIZEVAR>]
```

The OPEN statement is used to open a file channel number so the file can be accessed by a baZic program. The specified file is given the specified channel number. All references to this file are made via the channel number until the file is closed. When this command is executed, baZic internally defines a buffer region for transfer of information to and from the file.

The TYPEEXPR is an optional argument which can be passed to specify the type of the file to be opened. If no type expression is specified, baZic will assume a type 3 (data file). The TYPEEXPR allows a baZic program to open any file, including baZic programs and machine language programs. The OPEN will be successful only if the TYPEEXPR matches the actual type of the file.

The optional SIZEVARIABLE is passed to determine the size of the file. Upon a successful file OPEN, the SIZEVAR will contain the size of the file in **blocks**.

Only one file may be currently assigned to a channel number. That file must be CLOSED to free the channel number before any other file can claim it by the OPEN statement. Legal file channels are numbered 0 to 7.

An example of the OPEN statement follows:

```
OPEN #1,"DATAFILE"
10 OPEN #7%2,"MENU",S
20 OPEN #2,A$+D$
```

3.4.4 CLOSE a channel

```
CLOSE #<CHANNEL#>
```

The CLOSE statement is used to terminate a channel number so that the file previously associated with that channel number is no longer OPEN. The channel buffer is "flushed" so that all file data in RAM is written to the file before the file is CLOSED. The internal buffer space is now available when the specified channel number needs to be reused by another OPEN statement.

Channels are automatically CLOSED by baZic when any of the following conditions are met:

One program CHAINS to another program.

A program encounters an END or STOP statement.

A control C is entered by the user. (The buffer is flushed but the channel still remains "OPEN.")

The program terminates execution because of an error.

Examples of the CLOSE statement are as follows:

```
CLOSE #1
10 CLOSE #A
20 CLOSE #7
```

3.4.5 READ a file variable

```
READ #<CHANNEL#>[%<RANDOM ADDRESS>],<LIST OF VARIABLES>
```

The READ statement is used to READ variables from disk files into internal variables for use in a program. The specified CHANNEL# must have been previously OPENed for the READ statement to work. The CHANNEL# must be in the range of 0 to 7. A READ increments the file pointer to the byte following the variable read.

The optional RANDOM ADDRESS argument can be passed to read a file randomly. The value passed to this argument must be a positive integer in the range of 0 to the last byte of the file. The address passed is the offset from the beginning of the file to the position you want to read.

If any variable in the LIST OF VARIABLES argument is preceded by the ampersand sign (&), the variable will be "filled" with the byte value at the specified location. This value will be in the range of 0 to 255, the range of a byte value.

Examples of the use of the READ statement follow:

```
READ #1,A,B,C
10 READ #1%512*N,A,B,B$
20 READ #1,&A,&B,&C
```

3.4.6 WRITE a file variable

```
WRITE #<CHANNEL#>[%<RANDOM ADDRESS>],<LIST OF EXPRESSIONS>
```

The WRITE statement is used to write the specified LIST OF VARIABLES to the file associated with the CHANNEL#. The file must have been OPENED to the proper CHANNEL# before writing can take place. The CHANNEL# must evaluate to a positive integer in the range of 0 to 7. After a WRITE, the file pointer points to the byte after the last variable written.

The optional RANDOM ADDRESS can be specified for random writing of the file. This value is the offset (number of bytes) from the beginning of the file to the position to which you want to WRITE. The file pointer can be set to the beginning of the file by issuing a random write to position 0 followed by a NOENDMARK. The fourth example demonstrates this feature.

WRITEing begins at the current position of the file pointer and continues in a sequential fashion until all the variables specified have been written. An ENDMARK is written following the list unless the list specifies that no end mark is to be written by the NOENDMARK reserved word.

Variables can be written to the file in byte mode by the use of the ampersand sign (&) before each variable that is to be written bitwise. These variables must have a byte value in the range of 0 to 255.

Examples of the use of the WRITE statement are:

```
WRITE #1,A,B,C
10 WRITE #D%512*N,A,B,C,B$
20 WRITE #4%256,&A,&B,&C,NOENDMARK
30 WRITE #1%0,NOENDMARK           (Position pointer to beginning
                                   of file)
```

3.4.7 APPEND a program

```
APPEND [<LINE#>,<FILENAME>]
```

The APPEND statement is a special file statement. It is used only with type 2 files (baZic program files). This statement causes the specified program to be appended (added) to the program already in internal memory. The combination of the two programs now becomes the current program.

If the optional LINE# argument is not passed, the line numbers of the APPENDED program must be greater than the maximum line number of the program already in memory. If the LINE# argument is passed, all lines equal to or greater than the specified line number will be **replaced** by the lines in the specified program (FILENAME).

Upon the execution of the APPEND statement, all variables are cleared and processing resumes at the first statement of the following line. Multiple APPENDs are allowed, but each APPEND must be on a separate line. Nothing following an APPEND on a line is executed and therefore no statements should follow the APPEND on a line except a REMark statement. The only exception to this is if the APPEND statement is used in an IF THEN statement. In the following program, if A does not equal zero (A<>0), the GOTO will be executed.

```
10 IF A=0 THEN APPEND A$ ELSE GOTO 500
```

3.4.8 CHAIN to a program

```
CHAIN <FILENAME>
```

The CHAIN statement is another special file statement. The CHAIN statement causes the current program to be SCRatched from internal memory and the specified program to be LOAded into memory with an implied RUN command. All variables from the previous program are cleared and all files are automatically CLOSed.

The file name must be a legal program file name with a type of 2. The CHAIN statement is used so the operator of the program need not worry about LOAding programs or RUNning them.

Examples of the CHAIN statement follow:

```
CHAIN "MENU,2"
10 CHAIN A$+B$
```

3.4.9 DOSCoMmand

```
DOSCMD <ANY MICRODOZ COMMAND>
```

The DOSCMD statement is available only in the MicroDoZ version of baZic.

The DOSCMD statement is a very powerful statement allowing any MicroDoZ command to be executed from baZic. DOSCMD can be executed in the direct mode as a command or as a statement in a baZic program. The argument to DOSCMD can be any string expression that evaluates to a legal MicroDoZ command. If the command is invalid or cannot be executed, a trappable FILE error is returned. Multiple commands can be passed to MicroDoZ at one time as long as the total length of the commands is not greater than 127 bytes. Commands should be separated by a backslash (\).

This statement has no limitations on commands that can be passed, making this command capable of destroying any programs or data. This statement should be used with care since it is possible to issue a Read Disk (RD) command that would overlay baZic, MicroDoZ, or your program. Likewise, a WRite disk (WR) command could be used to completely "wipe out" a disk file of valuable information.

DOSCMD can be used to great benefit. DOSCMD makes initializing a floppy diskette from a baZic program very easy. Also, disk to disk transfers are made easy and fast through the use of the DOSCMD statement. By DIMensioning a string variable to a large number and then calling the ADDRess function to determine the RAM location of the string, a large buffer can be established in the middle of a baZic program. After the buffer is established, successive RD and WR commands can be issued to transfer information from one disk to another.

Consult the MicroDoZ manual for information on its commands.

3.5 Miscellaneous Statements

The following statements are general in nature and perform numerous tasks for the baZic programmer.

3.5.1 DIMension a variable

DIM <VARIABLE NAME> (<ARRAY OR STRING SIZE >)

The DIMension statement is used to allocate space needed for numeric array variables or string variables that will be longer than 10 bytes. Multiple DIMensions can be performed on the same program line by separating each variable by a comma.

A numeric array variable automatically is DIMensioned to 11 (elements 0 to 10) simply by using the variable. If a numeric array is to contain more than 11 elements, it must be DIMensioned to the proper number of elements to be stored within the array.

Multiple dimensioned arrays are allowed. Each numeric variable can be defined by the DIMension statement to as many dimensions as space in internal memory will allow. Each dimension within an array DIM statement must be separated by a comma. See the examples for multi-dimensioned arrays.

Strings under baZic may not be arrays as such. However, string arrays can easily be simulated under baZic. Strings can be any length, limited only by the amount of internal memory available. A string is automatically DIMensioned to 10 bytes (or characters) by simply using the string in a program before it is dimensioned.

Once a string or array is used in a program or DIMensioned, its dimension cannot be changed within that program. The argument passed to a DIMension statement for a string is the maximum number of bytes allowed in the string. All the positions in a string are set to ASCII spaces when a string is dimensioned.

Examples of the use of the DIM statement follow:

```
10 DIM A(20),B(20) (DIM numeric A and B to 21 elements)
20 DIM S$(200) (DIM S$ to 200 spaces)
30 DIM B(10,10) (DIM numeric B to 2-dimensional array
with 10 by 10 elements)
```

3.5.2 REMark a comment

```
REM <ANY LINE OF TEXT>
```

The REMark statement is one of the most valuable statements in baZic. This statement lets the programmer describe what he/she is doing without the description interfering with the program execution. When baZic "sees" a REM statement it takes no action and goes to the next line number to begin operation. No statements will be executed after a REM on a statement line because everything following a REM statement on a line is assumed to be part of the REMark field.

Examples of the REMark statement follow:

```
10 REM THIS IS A REMARK STATEMENT
20 A=20\REM A IS THE NUMBER OF RECORDS TO PRINT PER PAGE
30 REM A SHORT REMARK\X=20\REM X WILL NOT BE SET TO 20
```

3.5.3 LET (assign a variable)

```
[ LET ] <NUMERIC VARIABLE>=<NUMERIC EXPRESSION>
or [ LET ] <STRING VARIABLE>=<STRING EXPRESSION>
```

The LET statement is used to assign a value to a variable. Most programmers do not use the optional reserved word "LET." In any case, the value of the expression is assigned to the variable. Although LET can assign only one variable at a time, multiple LET statements are allowed on any program line.

When assigning string variables, full use of substrings is allowed. In other words, any part of any string can be assigned any part or all of any other string as long as there is enough room in the string variable.

Examples of the LET statement are:

```
LET A=5
B=75
10 LET A$=B$+C$+D$
20 B$(1,15)="FIRST NAME"
30 A=B*C+(X-Y/Z)
```

3.5.4 Clear the Screen

```
CLS [#<DEVICE#>]
```

The CLS statement is used to clear the screen of the CRT. This command is CRT-specific and cannot be used until MicroDoZ or baZic has been configured for your CRT. The program CONFIG is used to establish the clear screen sequence for MicroDoZbaZic and the program CRT is used for the CP/M and DOS versions. CLS can be used as a direct command or as a statement in a program.

Examples of the CLS statement follow:

```
CLS #1
100 CLS\REM CLEAR THE SCREEN
220 IF D=0 THEN CLS
```

3.5.5 FILL a memory location

```
FILL <LOCATION>,<BYTE VALUE>
```

The FILL statement is used to place a byte value directly in internal memory. Care must be taken in using this statement since baZic and MicroDoZ reside in internal memory and FILLs to locations within these programs can result in catastrophic system failures.

The LOCATION must be a positive integer Decimal address of a valid memory location (0 to 65535). The BYTE VALUE must be a positive integer in the range of 0 to 255.

Examples of the FILL statement follow:

```
FILL D2,6
10 FILL 12405,201
```

3.5.6 SET ERROR trapping

```
ERRSET [<LINE#>,<ERROR LINE NUMBER>,<ERROR NUMBER>]
```

The ERRSET statement is used to enable or disable the error trapping mode. If the ERRSET statement is used without the arguments, error trapping is disabled.

If the ERRSET statement is used in a program followed by the optional line number and two numeric variables and a trappable error occurs in the program, processing will branch to the specified line number and the first numeric variable will contain the line number where the error occurred and the second numeric will contain the number of the error that occurred. The number and meaning of the trappable errors can be found in Section 7.1 (Trappable Errors).

Once an error has occurred under ERRSET, the ERRSET statement must be executed again to re-enable the error setting mode. Control C can be trapped as an error if desired.

If the value of the line number is used in the recovery routine, it is the programmers responsibility to make the necessary changes after a RENumber. The RENumber command has no way of knowing the value of a variable in a program.

Examples of the ERRSET statement follow:

```
10 ERRSET 200,E1,E2
20 ERRSET
```

3.5.7 LINE length

```
LINE [#<DEVICE#>,<#EXPR>[,<#EXPR>]
```

The LINE statement is used to set the line length of any device. The default line length is 80 characters. The line length can be set for any value from 10 to 165 characters. The DEVICE# must be a legal device in your system and must evaluate to a positive integer from 0 to 7.

Normally, baZic echos a carriage return and line feed when the line length is reached while printing to a device number. If this is not desirable, the carriage return and line feed can be suppressed by passing an additional optional numeric expression which evaluates to zero (0). To re-enable output of the carriage return and line feed when the line length is reached, the optional numeric expression should evaluate to a non zero value.

When a LINE length is set, the value remains only through the current session with baZic. The LINE statement may be executed in a program or as a direct command.

Examples of the LINE statement follow:

```
LINE 132
10 LINE #2,88
```

3.5.8 STOP a program

```
STOP
```

The STOP statement causes a program to stop executing. The program may be CONTinued after a STOP is encountered by the CONT command. The program may not be modified during a STOP, but variables can be modified and printed to determine their values before CONTinuing.

Examples of the use of the STOP statement are:

```
10 STOP
20 IF N<>1 THEN STOP
```

3.5.9 END a program

END

The END statement is similar to the STOP statement except the END statement causes the program to terminate to the direct mode with no recourse to CONTINUE. END need not be the last line in baZic as with many other BASICs since baZic will assume an END statement when the last line of a program is executed.

The END statement may occur anywhere in a program and will not cause the program to "end" unless the END statement is executed. There may be any number of END statements in a program, or there may be none. There is an implied END statement at the end of the program.

Examples of the use of the END statement follow:

```
100 END
120 IF A$="END" THEN END
```

BUILT-IN FUNCTIONS

baZic contains numerous "built-in" functions to facilitate programming with the language. These functions provide easy access to many routines which are used frequently in the course of writing a program. The built in functions can be used for mathematical purposes, string manipulations, I/O, file manipulations, and other miscellaneous jobs.

In the examples of functions that follow, the result of the function call will appear to the right of the function example and will be enclosed in parenthesis.

Numeric functions may be freely used in all numeric expressions. Functions may also be used in FOR NEXT loops or IF THEN ELSE statements. Several examples of the general use of functions follow:

```

10 IF COS(A)=INT(B) THEN PRINT Sqrt(C) ELSE PRINT LOG(D)
20 FOR N=1 TO ABS(X)\NEXT N
30 ON ABS(INT(Y)) GOSUB 40,50,60,70
40 IF LEN(A$)>32 THEN 500
50 ON VAL(A$) GOTO 10,20,30,40
60 PRINT ABS(C)

```

Many of the numeric functions can be removed from baZic by the program SHORTB to reclaim additional programming space.

4.1 Math Functions

The following functions are all related to mathematical calculations.

4.1.1 ABSolute value

ABS(<NUMERIC EXPRESSION>)

The ABSolute value of an expression is the positive result of the numeric expression without regard to the sign. All positive numbers remain positive and all negative numbers are made positive.

Examples of the use of the ABS function follow:

A=ABS(10)	(A=10)
B=ABS(-15)	(B=15)
C=ABS(0)	(C=0)

4.1.2 SiGN of a number

SGN(<NUMERIC EXPRESSION>)

The SiGN function is used to determine the sign of a number (i.e., is it positive or negative?). The function accomplishes this by returning a +1 if the expression is positive, -1 if the expression is negative, and a 0 if the value of the expression is zero (0).

Examples of the SGN function follow:

A=SGN(0)	(A=0)
B=SGN(577*21)	(B=1)
C=SGN(43-55)	(C=-1)

4.1.3 INTeger value

INT(<NUMERIC EXPRESSION>)

The INTeger function is used to return only the integer value which is less than or equal to the value of the numeric expression. Any fractional part of the number is discarded. Notice that this function does not round off a number, it only returns the integer part of the expression.

Examples of the INT function are:

A=INT(0)	(A=0)
B=INT(14.9455)	(B=14)
C=INT(-6.5367)	(C=-7)

4.1.4 LOGarithmic value

LOG(<NUMERIC EXPRESSION>)

This function returns an approximation (based on the precision of baZic) to the natural logarithm of the value passed as a numeric expression. The value of the argument must always be positive (greater than 0).

Examples of the LOG function are:

A=LOG(1)	(A=0)
B=LOG(.5)	(B=-.69314717)
C=LOG(23.14069)	(C=2.1415925)

4.1.5 EXPonential value

EXP(<NUMERIC EXPRESSION>)

This function returns an approximation (based on the precision of baZic) to the value of e raised to the power of the numeric expression.

Examples of the EXP function are:

```
A=EXP(0)           (A=1)
B=EXP(-.69314717) (B=.5)
C=EXP(3.1415925)  (C=23.14069)
```

4.1.6 Square Root

SQRT(<NUMERIC EXPRESSION>)

This function returns an approximation of the square root of the numeric expression. The argument to this function must be greater than or equal to zero (0). NOTE: Many BASIC interpreters use "SQR" for the square root function instead of "SQRT" as in baZic.

Examples of the SQRT function are:

```
A=SQRT(0)           (A=0)
B=SQRT(4)           (B=2)
C=SQRT(3)           (C=1.7320508)
```

4.1.7 SINE

SIN(<NUMERIC EXPRESSION>)

This function returns an approximation of the trigonometric sine of the value passed as the numeric expression. The expression must pass the angle in radians.

Examples of the SIN function are:

```
0=SIN(0)
1=SIN(3.1415926/2)
```

4.1.8 COSine

COS(<NUMERIC EXPRESSION>)

This function returns an approximation of the trigonometric cosine of the value passed as the numeric expression. The expression must pass the angle argument in radians.

Examples of the COS function are:

```
A=COS(0)           (A=1)
B=COS(3.1415926)  (B=-1)
```

4.1.9 ArcTaNgent

ATN(<NUMERIC EXPRESSION>)

This function returns an approximation of the trigonometric arc-tangent of the value passed as the numeric expression. The value returned is an angle expressed in radians.

Examples of the ATN function are:

A=ATN(3)	(A=1.2490457)
B=ATN(.75)	(B=.6435011)

4.2 String Functions

The following functions are designed to facilitate the use of strings by providing functions that tell the LENgth of strings, functions that convert string information to numeric and vice versa and functions that convert string information to their ASCII equivalent and back again.

4.2.1 LENgth of a string

LEN(<STRING NAME>)

The LENgth function is designed to return the length of the specified string. The length of a string is the number of characters within the string. The length of a null string is zero (0).

Examples of the LEN function follow:

```

10 DIM A$(20),B$(20),C$(20)
20 A$="MICRO MIKE'S"
30 B$=""
40 PRINT "THE LENGTH OF A$ IS",LEN(A$)
50 PRINT "THE LENGTH OF B$ IS",LEN(B$)
60 PRINT "THE LENGTH OF C$ IS",LEN(C$)

```

When this program is RUN the results will be:

```

READY
RUN

THE LENGTH OF A$ IS 12
THE LENGTH OF B$ IS 0
THE LENGTH OF C$ IS 20
READY

```

4.2.2 CHaRacter\$

CHR\$(**<NUMERIC EXPRESSION>**)

The CHaRacter String function is passed the decimal ASCII value of a character and the function returns a one-character string that represents the selected ASCII character. The range of arguments for this function is from 0 to 255. See APPENDIX C for the value of all ASCII characters.

Examples of the use of the CHR\$ function follow:

```
A$=CHR$(33)      (A$="!")
B$=CHR$(49)      (B$="1")
C$=CHR$(65)      (C$="A")
D$=CHR$(122)     (D$="z")
```

4.2.3 ASCii value

ASC(**<STRING EXPRESSION>**)

The ASCii function is the inverse of the CHR\$ function. The ASC function returns the ASCII value of a the first character of the specified string. The null string is not a valid argument for this function. The specified string can be a substring of a larger string.

Examples of the ASC function are:

```
A=ASC(" ")      (A=32)
B=ASC("B")      (B=66)
C=ASC("BOB")    (C=66)

10 A$="ABC"
20 A=ASC(A$(2)) (A=66)
```

4.2.4 VALue

VAL(**<STRING EXPRESSION>**)

The VALue function is used to convert a numeral in a string to a numeric variable. This function allows numbers to be entered into a string variable and then converted into a numeric variable. Leading blanks (ASCII spaces) are ignored. If the specified character(s) are not legal numeric constants, an error will be returned. Non-numeric values are allowed after the numeric values.

Examples of the VAL function follow:

```
A=VAL("00STRING") (A=0)
B=VAL("10")        (B=10)
```

4.2.5 STRing\$

STR\$(**<NUMERIC EXPRESSION>**)

The STRing\$ function is the inverse of the VALue function. The STR\$ function returns a string which corresponds to the numeric argument that is passed. The format of the returned string is dependent on the current default format. If the current default format is the free format, a space will always be inserted in the string as the first character of the number.

Examples of the STR\$ function are:

A\$=STR\$(1234) (A\$=" 1234")

If the current print formatting default is %#10F2, the preceding example would convert as follows:

B\$=STR\$(1234) (B\$=" 1234.00")

4.3 Input Functions

The input functions are involved with inputting a character or a byte from an input device.

4.3.1 INPut a CHARacter\$

INCHAR\$(**<DEVICE#>**)

This function takes in one character from the specified device number. The returned value is a single character string. This function does **not** echo the input character back to the input device. Control characters are allowed, but Control C can only be returned when control C is inhibited.

An example of the INCHAR\$ function follows:

A\$=INCHAR\$(0) (Input a character from device # 0)

4.3.2 INPut a byte

INP(**<PORT NUMBER>**)

This input function takes one byte from the specified Z80 port and returns this byte as a numeric value from 0 to 255. This function is equivalent to the Z80 IN instruction. The INP function does not validate the data it receives, but merely "grabs" whatever value is at the specified port at the time of the function call. The port number argument must be in Decimal. The value INPut is not echoed to the input device.

An example of the INP function follows:

I=INP(6) (Input a byte from port 6)

4.3.3 INput the STATus

INSTAT(<DEVICE#>)

The INSTAT function is available only in the MicroDoZ and CP/M versions of baZic.

The INSTAT function is used to determine the status of an input device to determine if the device is ready to send another character. If the value of INSTAT is one (1), the port is ready to send information. If the value of INSTAT is zero (0), the port is not ready to send information.

This function can be used to allow programs to be executing while waiting for the user to input information. The INSTAT function must be called often enough by the program to insure that all user keystrokes are "caught."

4.3.4 OUTSTAtus

OUTSTAT(<DEVICE#>)

The INSTAT function is available only in the MicroDoZ and CP/M versions of baZic.

The OUTSTAT function is similar to INSTAT except the OUTSTAT function is used to determine the status of an output device. If the device is ready to accept information, an OUTSTAT call will be equal to one (1). If the device is not ready to accept information, the status of the device will be zero (0).

This function can be used to print information and do other processing simultaneously. If OUTSTAT is not called often enough the printer will appear to slow down.

4.4 File Functions

The following functions are used to return useful information about the files and associated file pointers.

4.4.1 TYPE of file pointer

TYP(<CHANNEL#>)

The TYPE function returns the type of data currently being "pointed to" by the file pointer of the specified CHANNEL#. The CHANNEL# must have been previously OPENED by an OPEN statement. If the function returns a "0", the next item in the file is an END MARK. If the function returns a "1", the next item in the file is a string. If the function returns a "2", the next item in the file is a numeric. This function has no provisions for byte values.

An example of the TYP function follows:

```
A=TYP(2)           (Where 2 is a previously OPENed file)
```

The previous discussion assumes a sequential file. With random or byte level files, the file pointer can be pointing at a 0, 1, or 2 by coincidence and not be pointing at an endmark, string, or number.

4.4.2 FILE type

```
FILE(<FILENAME>)
```

The FILE function returns the type of the specified file. The FILENAME argument must evaluate to a legal file name. The type information is returned as a numeric value. If the file does not exist, the function returns a -1. If the file does exist, the type of the file is returned. A machine language file is generally a type 1, a baZic program is type 2, and a baZic data file is generally a type 3. Values for the type function can range from 0 to 127.

Examples of the FILE function follow:

```
A=FILE("BAZIC")
B=FILE("DATAFILE")
C=FILE(A$)
```

4.4.3 FILESIZE

```
FILESIZE(<CHANNEL#>)
```

The FILESIZE function is used to return the size of the specified CHANNEL#. The channel must have previously been opened by an OPEN statement. The value returned will be the size of the file in blocks.

An example of the FILESIZE function follows:

```
A=FILESIZE(2)      (Return the size of the channel 2 file)
```

4.4.4 FILEPoinTeR position

```
FILEPTR(<CHANNEL#>)
```

The FILEPoinTeR function causes the position of the file pointer in the specified CHANNEL# to be returned. The CHANNEL# must have been previously opened with an OPEN statement.

An example of the FILEPTR function follows:

```
A=FILEPTR(3)      (Return the pointer position of channel 3)
```

4.5 Miscellaneous Functions

The following functions are unrelated and perform different tasks. Functions performed include generating RaNDom numbers, EXAMining and CALLing internal memory locations, positioning the cursor by TABulating, determining the amount of FREE memory, and returning the RAM ADDRESS of a variable.

4.5.1 RaNDom

RND(<#EXPR>)

The RaNDom function returns a psuedo-random value between 0 and 1. The argument passed as a #EXPR is called the seed and determines which psuedo-random sequence is to be generated. A random seed can be generated by baZic by passing a negative one (-1) as the argument. When a negative one is passed as the argument, baZic uses the value of the Z80 refresh register for the seed.

If the NUMERIC EXPRESSION evaluates to a zero (0), the previous seed is used so that the next number in that particular random sequence is generated. In this way the same "random" sequence can be duplicated by calling the RND function first with a seed value and then all successive calls with a "0" argument.

Examples of the RND function follow:

```
A=RND(-1)           (baZic should "pick" the seed)
B=RND(.0998)        (seed is .0998)
C=RND(0)           (use previous number to generate next #)
```

4.5.2 EXAMine memory

EXAM(<MEMORY LOCATION>)

The EXAMine function is used to allow baZic programs to "look" at specific bytes in internal memory. The MEMORY LOCATION argument must be a positive integer between 0 and 65535 and must be a Decimal number. The value returned from the function call will be a byte value in the range of 0 to 255.

Examples of the EXAM function follow:

```
10 A=EXAM(12405)
20 IF EXAM(12405)=201 THEN D2=12405
30 PRINT EXAM(N)
```

4.5.3 FREE memory

```
FREE(<DUMMY ARGUMENT>)
```

The FREE memory function is used to return the amount of internal memory that is free within baZic. This is memory not used for baZic itself, the current program, the used and dimensioned variables, and the data storage of the baZic program. The DUMMY ARGUMENT is not evaluated by the function and any numeric value will work.

An example of the FREE function follows:

```
PRINT FREE(0)
```

4.5.4 TABulate

```
TAB(<#EXPR>)
```

The TAB function is used in PRINT statements to cause the cursor or print head to be positioned to the value of the #EXPRESSION argument. The TAB position is always the absolute position and not a relative position calculated from any previous position. No value is returned from the function but the print head or cursor is positioned to the argument value. No action is taken by the function if the argument is before the current position of the cursor or print head. TABs may be used after a PRINT@ statement since the PRINT@ statement always updates the print head position table.

Examples of the TAB function follow:

```
10 PRINT TAB(20),"THIS IS POSITION 20"  
20 IF X<>Y THEN !TAB(10),"X" ELSE !TAB(20),"Y"
```

4.5.5 CALL machine language

```
CALL(<MEMORY ADDRESS> [,<DE ARGUMENT>])
```

The CALL machine language function is designed to allow baZic programs to interface with machine language subroutines. This feature can be very desirable since many routines in machine language operate as much as 100 times faster than the same routine in baZic or any other BASIC interpreter.

The MEMORY ADDRESS argument is a positive Decimal integer in the range of 0 to 65535. This value is the address of the routine to be CALLED. The second optional DE ARGUMENT is a value in the same range that will be placed in the DE register pair to pass a value to the machine language routine. The CALL function returns the value of the HL register pair.

To terminate the machine language routine, execute a RETURN instruction. All registers except the IX and IY can be used by the machine language routine; however the stack and stack pointer should **not** be modified. If stack operations are required, the stack pointer should be saved and a new stack established. The IX and IY can be used if they are returned to baZic unmodified. All other registers can be freely modified by the user's routines.

Examples of the CALL function follow:

```
A=CALL(63455)
B=CALL(A,H)
```

4.5.6 ADDRESS of a variable

```
ADDR(<VARIABLE NAME>)
```

The ADDRESS function is used to return the address of a variable within the current baZic program. If the specified variable is a string, the function will return a decimal number that points to the RAM address of the first byte of the string. If the argument to the function call is a numeric variable, the function will return the address of the exponent (last) byte of the number.

An example of the ADDR function is as follows:

```
A=ADDR(A$)      (A = the RAM address of first byte of A$)
B=ADDR(C)       (B = the RAM address of the exponent of C)
```

OPERATORS

Operators in baZic are special signs, characters, or words that cause one or more numeric values (operands) to be changed by the operation. The operators in baZic are classified as arithmetic, relational, and logical.

Any combination of numeric constants, numeric variables, operators, function calls, and array variables can be considered to be a numeric expression. This feature gives tremendous flexibility in programming complex situations.

5.1 Arithmetic Operators

The arithmetic operators are the common operators seen in everyday arithmetic. Some special symbols must be defined for use in baZic because most CRTs do not have a multiplication sign and have to use the asterisk (*) to signify multiplication.

The arithmetic operators are defined in the following table:

OPERATOR	FUNCTION	EXAMPLE
^	EXPONENTIATION	64=8^2
*	MULTIPLICATION	50=5*10
/	DIVISION	10=50/5
-	SUBTRACTION	12=25-13
+	ADDITION	15=5+10
-	NEGATION	-5=-(+5)

5.2 Relational Operators

The relational operators are the true/false operators. The result of a relational comparison is either true (=1) or false (=0). The relational operators are mainly used in the IF THEN ELSE statement to determine a branching condition.

The relational operators are defined in the following table:

OPERATOR	RELATION	EXAMPLES	
		TRUE (1)	FALSE (0)
=	EQUAL	1=1	1=2
<	LESS THAN	1<2	1<1
>	GREATER THAN	2>1	2>2
<=	LESS THAN OR EQUAL	2<=2	2<=1
>=	GREATER THAN OR EQUAL	2>=1	2>=3
<>	NOT EQUAL	2<>1	2<>2

5.3 Boolean Operators

The three Boolean operators are AND, OR, and NOT. These operators may be combined with the relational and arithmetic operators to handle more complex situations. All non zero values are considered true while zero values are considered false.

For the AND condition to be true, both the first operand AND the second operand must be true. The following program evaluates the AND condition as true AND false:

```

10 X=1
20 Y=5
30 IF X=1 AND Y=5 THEN (THE EXPRESSION IS TRUE)
40 IF X=1 AND Y=3 THEN (THE EXPRESSION IS FALSE)

```

For the OR condition to be true, one operand OR the other must evaluate to be true. Only if both operands are false is the OR operation false. The following program demonstrates the OR condition.

```

10 X=1
20 Y=5
30 IF X=1 OR Y=5 THEN (THE EXPRESSION IS TRUE)
40 IF X=1 OR Y=1 THEN (THE EXPRESSION IS TRUE)
50 IF X=0 OR Y=5 THEN (THE EXPRESSION IS TRUE)
60 IF X=0 OR Y=1 THEN (THE EXPRESSION IS FALSE)

```

The NOT condition is true when the operand is NOT false. The NOT operator negates the Boolean value of an operation. If <OPERAND> is false, NOT<OPERAND> is true. The following program demonstrates the NOT condition:

```

10 X=1
20 Y=5
30 IF NOT (X>Y) THEN (THE EXPRESSION IS TRUE)
40 IF NOT (X<Y) THEN (THE EXPRESSION IS FALSE)

```

Boolean expression may also be "formed" by the use of numeric variables and constants. The following sets of programs will demonstrate the use of several abstract Boolean expressions:

```

10 X=1
20 Y=5
30 Z=X=Y

```

Since X does not equal Y the value of Z upon executing line 30 will be zero (false). If X and Y were equal, Z would be set to 1 (true).

```

10 X=3
20 Z=X>5
30 A=NOT Z

```

In this example, Z is set to 1 (true) if X is greater than 5, otherwise Z will be set to 0 (false). The variable A will be the complement of Z.

```
1 0 N= (X=Y) = (Z=W)
```

This example shows a hybrid logical expression. The expression is evaluated such that N will be set to 1 (true) when X=Y and Z=W. If X<>Y or Z<>W, N will be set to 0 (false).

The expression "IF NOT X" is equivalent to "IF X=0" and uses one less byte of program storage. In a similar manner, "IF X" is equivalent to "IF X<>0" and save three bytes of program storage.

5.4 Order of Evaluation

To avoid confusion in the evaluation of arithmetic, relational and boolean operators, baZic defines the precedence of the operators. This precedence may be changed by the use of parentheses. The higher precedence operators are always evaluated first and operators of equal precedence are evaluated from left to right. Operators enclosed in parentheses are evaluated before operators not enclosed in parentheses. Parentheses pairs can be nested within other parentheses but the innermost pair is evaluated first.

The following table lists the precedence of the operators starting with those of highest precedence. All operators listed on the same line are of equal precedence.

NOT, -	(logical operator, negate a number)
^	(exponentiation)
*,/	(multiplication and division)
+,-	(addition and subtraction)
=,<,>,<=,>=,<>	(relational operators)
AND	(logical operator)
OR	(logical operator)

USER-DEFINED FUNCTIONS

baZic has the ability to let the user define his own functions. All of the built-in functions could be defined in baZic but their operations would be much slower. This ability greatly enhances the usefulness of baZic. User-defined functions are "handy" because they can be defined at any place in the program and can be called from any place in the program without regard to line numbers. User-defined functions, in general, do not execute as fast as a subroutine which performs the same function.

User-defined functions may be defined to return one numeric or string value. Functions are named the same as variables except a function has "FN" followed by the name. If the function name is a string name, the function is a string function. As many arguments as desired can be passed to user defined functions and these can be either string or numeric but only one value is returned.

A user function is defined by the reserved word "DEF" which "tells" baZic the user is defining a function. Functions can be one line or as many as required. The definition will include variables enclosed in parentheses to represent the arguments to be passed to the function. These variables defined in the parameter list of a function DEFinition become local variables to the function itself.

All numeric variables used within a function DEFinition and declared in the function definition will be local to the function. Changing these variables within the function will not effect variables of the same name used elsewhere in the program. All numeric variables not declared in the function DEFinition are global variables and any use of these variables in the function will affect and change the value of the variable elsewhere in the program.

String variables are always global and the use of string variables in the DEFinition of the function will not change their global nature.

baZic allows single and multiple line function DEFinitions. A single line user defined function must have the entire function defined on a single line. The single line function DEFinition appears as follows:

```
DEF FNA(R,C)=!@(R,C)
```

Multiple line function DEFINitions contain only the DEFINition reserved word, the function name, and the parameter list. The same function defined in the single line example defined as a multiple line function would appear as follows:

```
10 DEF FNA(R,C)
20 !@(R,C)
30 RETURN R
40 FNEND
```

6.1 DEFine a function

```
DEF <FUNCTION NAME> (<PARAMETER LIST>) [=<EXPRESSION>]
```

The DEF statement is used to inform baZic that the user is defining a function. If the function is to be a single line function, the =EXPRESSION must be included in the definition. If the function is to be a multiple line function then the =EXPRESSION is omitted. All user defined functions are defined at RUN time before the actual program execution begins.

Examples of the use of the DEF statement appear in the previous section.

6.2 RETURN from a function

```
RETURN <NUMERIC OR STRING VARIABLE>
```

The RETURN statement is used to terminate a function and RETURN a variable to the calling program. The value assigned in the RETURN statement is the value that will be returned from the function call. If the function is a numeric function, the value RETURNed from the function call must be numeric and if the function is a string function, a string variable must be returned.

A function may contain more than one RETURN statement. The first RETURN executed terminates the function call. A RETURN also "EXITS" all FOR NEXT loops within the function call. See Section 3.3.7 for more information.

Examples of the use of the RETURN statement follow:

```
100 RETURN A
200 RETURN B$
```

6.3 FuNctionEND

The FuNctionEND statement is used to signify the end of each multiple line function defined. If the function doesn't have a FNEND statement, an error will be generated.

An example of the FNEND statement can be found in Section 6 (USER DEFINED FUNCTIONS).

ERROR MESSAGES

This section is designed to detail the error messages which can be returned by baZic. The error messages are divided into two sections: the trappable errors and the non-trappable. Many of the errors listed are followed by the statements, commands, or internal routines which cause the error. Internal routines are assembly language routines. These routines are named with symbolic names and shouldn't necessarily make sense to a reader of this manual. The names are provided so that the user can gain insight into the conditions which will cause an error to be generated.

7.1 Trappable Errors

Trappable errors are errors which can be trapped using the ERRSET statement of baZic. These errors are generally ones that are not catastrophic and the programmer usually has some recourse when these errors occur. The errors are listed in the sequence of their error number to facilitate finding the error when the error number is known.

7.1.1 ARGument (Error 1)

The ARGument error is returned anytime a function, command, or statement is given an invalid argument. All functions can return this error as well as the following commands, statements, and internal routines:

LIST	Improper command format.
MEMSET	Not followed by a number. No memory at that address.
DEL	Not followed by a number. Second line number less than the first.
LOAD	DLOOK failure - Normally no file by that name.
NSAVE	File name in use. Size specification is not a number.
SAVE	DLOOK failure - Normally no file by that name.
LN2LC	This is a subroutine used by RUN, LIST, EDIT, and DEL. It converts a line number in the command buffer to an address. An error occurs if the argument is not a number.

REN	Improper command format.
AUTO	Improper command format.
APPEND	DLOOK failure - Normally no file by that name.
CHAIN	DLOOK failure - Normally no file by that name.

7.1.2 DIMENSION (Error 2)

The DIMENSION error refers to a problem using the DIM statement. Either the programmer has tried to re-dimension a numeric array or string variable, or the programmer has tried to DIMENSION a numeric array or string variable after the variable has been used in the program. The following statement can cause this error:

DIM	DIM is not followed by a variable.
	Zero dimension for a string.
	Same variable dimensioned twice.

7.1.3 OUT OF BOUNDS (Error 3)

The OUT OF BOUNDS error can be returned on most functions and many commands and statements. This error can be caused when a numeric argument is not within the prescribed range or the programmer makes a reference to a numeric array or string variable that is outside the limits of the variable. The error will also occur when the programmer attempts to READ or WRITE beyond the limits of a disk file.

The following commands, statements, functions, and internal routines can generate an OUT OF BOUNDS error:

MEMSET	Specified value will truncate current program.
AUTO	Line number increment is zero.
REN	Line number increment is zero.
DEL	Line numbers not in range of 0 to 65535
SAVE	Not enough space on disk or program larger than file.
ON	Argument greater than 255.
TAB	Argument greater than 255.
OUT	Port and/or value greater than 255.

- WRITE Attempting to write a byte value greater than 255. Not enough space in file to complete this operation.
- FLSTAT This subroutine locates the status byte for the specified file. An error results if the specified file number is greater than 7.
- FCOM An initialization routine used by READ# and WRITE# prior to calling the DOS DCOM routine. An error is given if the file pointer value is greater than the number of blocks in the file.
- LINE Attempting to set the line length to a value greater than 165 or less than 10. Device number greater than 7.
- CREATE Specified file type greater than 127.
- ASC Argument is a null string.
- INP Port greater than 255.
- LVR A subroutine used to locate variables. An error occurs when an attempt is made to locate an element of an array which is outside the specified dimensions.
- Attempting to access string element zero.
- For a partial string if the specified end is actually before the specified beginning.
- REN RENUMbering would result in a line number greater than or equal to 65535.
- ATOBI A subroutine which converts ASCII numbers to binary. Any attempt to convert a number greater than 65535 gives an error.
- BCDA A subroutine which converts binary coded decimal to formatted ASCII. This routine automatically rounds if necessary. When the E format is used rounding may result in a number which is larger than the maximum; ie. 9.9999999E+62 for 8-digit precision.
- EXTBI A subroutine used to convert numerical expressions to file pointers. An error occurs if the block pointer exceeds 65535.
- A error occurs if the pointer is negative.

BIMPY	A subroutine for binary multiplication. Used by DIM, FLSTAT, LVR, and another subroutine which allocates memory space for arrays. An error is given if the result will exceed 65535.
SQRT	Attempting to take the square root of a negative number.
EXP	Argument greater than 145.062
LOG	Zero argument Argument less than 3.1622776E-64. Argument less than zero.
SETPO	Any attempt to set print device greater than 7.

7.1.4 TYPE (Error 4)

The TYPE error is returned when the programmer tries to assign a numeric value to a string variable or assign a string value to a numeric variable. The error also occurs when an attempt is made to OPEN a file whose type does not agree with the type specified in the program.

The following can cause a TYPE error:

OPEN	File type specified is not the same as the file found. Block variable is a string variable.
SAVE	Not a type 2 file.
LOAD	Not a type 2 file.
EXCFN	A routine used to initialize FN execution. An error occurs if a string function is used in a numerical expression or vice versa.
EVSTR	A routine used to evaluate string expressions. An error occurs if a numerical variable is used in a string expression.
LEN	Argument is a variable but not a string variable.
ASC	Argument is a variable but not a string variable.
EVLN	Routine used to evaluate numeric expressions. An error occurs if the expression contains a string variable.

ERRSET The variable for error type is a string variable.
 The variable for line number is a string variable.

READ# Attempting to read a single byte to a string.
 First string overhead byte neither 2 nor 3.
 High nybble of BCD number is zero.

FOR Index variable is a string variable.

7.1.5 FORMAT (Error 5)

The FORMAT error is associated only with PRINT statements. The error occurs when the format parameters are not legally defined and when attempts are made to print a variable whose number of characters exceed the format length. The error can also occur when numbers containing fractions are printed in the I format.

The following statement and internal routine can cause the FORMAT error:

FORMAT Field greater than 33.
 Not I, E, A, or F.
 For E and F types
 Digits right of decimal not specified.
 Digits right of decimal greater than or
 equal the field less one.

BCDA I format specified for non-integer value.
 Field too small.

7.1.6 LINE NUMBER (Error 6)

The LINE NUMBER error is generated when a branching statement makes a reference to a line number that doesn't exist in the current program, or the reference to the line number contains some error such that a valid line number is not given as the argument to the statement.

The following commands, statements, and internal routines can cause a LINE NUMBER error:

FNDLN5 A routine which finds the address for a given line number. Used by GOTO, GOSUB, and RESTORE. Gives an error if the line number does not exist.

APPEND The first line number of the appended program is less than the last line number of the original program.

LIST	Line number does not exist.
DEL	Line number does not exist.
RUN	Line number does not exist.
ERRSET	Line number does not exist.
EXIT	Line number does not exist.

7.1.7 FILE (Error 7)

FILE errors are always related to disk files. The error occurs when the program tries to access a disk file that doesn't exist or is of the wrong type. If the CHANNEL NUMBER is not within the legal range or an attempt is made to OPEN a CHANNEL NUMBER already OPENed, a FILE ERROR will result. Attempts to CREATE or NSAVE files that will not fit in the remaining space on a disk or attempts to change any information on a write protected disk will cause the FILE ERROR message to be generated.

The following commands, statements, functions and internal routines can cause the FILE error to occur:

TYP	File not open.
NSAVE	Directory full. Not enough space.
CREATE	Directory full. Not enough space. File already exists.
DOS	Error detection in DOS.
FCOM	Unsuccessful DCOM.
READ#	File not open.
WRITE#	File not open.
OPEN	File not found. File already OPEN.
SAVE	Unsuccessful DCOM.
LOAD	Unsuccessful DCOM. File has zero sectors of valid data.

CAT	File not found.
DESTROY	File not found.
APPEND	File not found.
CHAIN	File not found.

7.1.8 HARD DISK (Error 8)

The HARD DISK error always is associated with the disk drives. This error can be caused by an improperly seated diskette or a diskette formatted incorrectly or for some reason has unreadable data in floppy disk systems. Hard disk systems should not return a HARD DISK error unless there is a hardware problem or all the substitution sectors on the hard disk have been used.

All error detection for the HARD DISK ERROR occurs in the DOS. The following commands and statements can generate a HARD DISK error:

CAT	SAVE	LOAD	APPEND	CREATE
DESTROY	CHAIN			

7.1.9 DIVIDE by ZERO (Error 9)

The DIVIDE by ZERO error can occur only in conjunction with the LET (or implied LET) statement or a numeric function. This error occurs when an attempt is made to divide by zero. The following internal routine can generate a DIVIDE ZERO error:

FPDIV	Attempt floating point divide by zero. (Software floating point version only. With the Hardware floating point version all errors are reported as numeric overflow)
-------	---

7.1.10 SYNTAX (Error 10)

The SYNTAX error is the most prevalent error to occur under baZic. Every function, statement, and command can cause a SYNTAX error if the function, statement, or command is not spelled correctly or if the attempted use is not in accord with the specifications of the function, statement, or command.

All commands, statements, and functions can generate this error but the following statements generate the error in ways that are often not obvious to the programmer:

DATA	Trying to READ a numeric constant into a string variable
ON GOTO	On variable larger than number of line numbers
ON GOSUB	On variable larger than number of line numbers
RETURN	No return variable (from a user defined function)
READ	Trying to READ a numeric constant into a string variable

The following statements and internal routines can generate a SYNTAX error:

PRSTR	The print string routine. Finds 0DH (carriage return) before a double quote.
EXCFN	Missing right parenthesis.
\$CONST	String constant. Finds 0DH before a double quote.
VFYB	Verify that next token is equivalent to value in Register B. Most frequent use is to verify expected left and right parenthesis and commas.
CHEKES1	A routine which checks end of statement. Gives error on improper ending.
LVR	Expecting a variable but didn't find one.
CHK\$	A routine which checks for string variable. Error if it doesn't find a variable of any type.
EVSTR	String constant expected but not found.
EVLN	Expression ends with an op code. Expression begins with a non-unary op code. Adjacent non-unary op codes.
ERRSET	No line number or one or both variables missing.
IF	With strings the only valid operators are >=, <=, <>, <, >, and =. Any other operator gives an error. The boolean expression includes a string constant without double quote at end.
ON	No line number after comma. Not GOTO or GOSUB
GOSUB	No line number.

GOTO	No line number.
EDIT	No line number.
PROP	Parses primary opcodes. Error given when a direct command is used in a program.
READY	Initialization routine. Error occurs when a statement doesn't end properly.

7.1.11 READ (Error 11)

The READ error only occurs with the READ statement. If the programmer attempts to READ a string constant into a numeric variable, the error will occur. Also, if the READ statement attempts to read beyond the available DATA statements or there are no DATA statements, a READ error will occur.

7.1.12 INPUT (Error 12)

The INPUT error occurs only with the INPUT or INPUT1 statement or the VAL function. If the programmer is "asking" for a numeric input and the user enters a string or any characters not constituting a numeric value, the error is returned. In this case baZic asks the user to RETYPE the input.

If the argument to the VAL function is not numeric, an INPUT error will occur. This error is trappable by ERRSET as an INPUT error.

7.1.13 ARGument MISMATCH (Error 13)

The ARGument MISMATCH error occurs when an attempt is made to access a user defined function but the number of arguments passed does not agree with the parameter list established in the function DEFINITION.

7.1.14 NUMERIC Overflow (Error 14)

This error is returned when attempts are made to exceed the range of the precision of baZic being used. If the precision of the baZic is 8, numbers that are greater than 9.9999999E+62 are not allowed. Numbers smaller than 1E-64 are converted to zero (0). This error is reported for all hardware floating point board arithmetic errors and all software floating point errors except an attempt to divide by zero.

7.1.15 STOP/Control C (Error 15)

This "error" occurs when Control C is enabled and error trapping is set by the ERRSET statement and the user presses the control C keys on the keyboard. This error is used to trap the control C input so the programmer can change program flow based on the user input.

7.1.16 LENGTH (Error 16)

The LENGTH error is returned when using the INPUT or INPUT1 statements or when the programmer is entering a program in the direct mode. This error occurs when the number of characters input is longer than the current line length. The line length can be set using the LINE statement to any value up to 165 characters.

7.2 Non-trappable Errors

Non-trappable errors are usually catastrophic in nature and preclude any chance of continuing the program. For this reason these errors cannot be trapped using the ERRSET command. These errors are generally gross programming errors where the programmer's logic has broken down.

7.2.1 CONTINUE

This error occurs when an illegal attempt is made to issue the CONT command to continue program execution. The program may not be CONTinued if a program error caused the program to terminate, or an END statement was encountered by the program, or the program has been changed since it was stopped. The program can be CONTinued if a control C was used to stop the program or a STOP statement was encountered in the program and no program lines have been changed.

7.2.2 CONTROL STACK

The control stack is used to store the loop counts on FOR NEXT loops and to store the RETURN addresses for subroutines and user defined functions. The improper use of any of these statements can cause a CONTROL STACK error to occur.

The following statements can cause a CONTROL STACK error:

FOR	The statement containing FOR is the last statement of the program.
NEXT	FOR-NEXT data not at top of control stack as expected. Wrong index variable.
EXIT	FOR-NEXT data not at top of control stack.
RETURN	No return data in control stack.

7.2.3 DOUBLE DEFINITION

The DOUBLE DEFINITION error always occurs at RUN time when all user defined functions are evaluated. This error is returned when more than one user defined function is found to have the same name.

7.2.4 FUNCTION DEFINITION

This error occurs in user defined functions when a new DEF statement is encountered before baZic can find the FNEND statement of the preceding function. This error is also generated when the programmer tries to access an undefined user function.

7.2.5 ILLEGAL DIRECT

This error is returned when an attempt is made to use illegally a statement or function in the direct mode. User defined functions cannot be used in the direct mode. See Section 2 (DIRECT COMMANDS) for a list of statements that can be used in the direct mode.

7.2.6 INTERNAL STACK OVERFLOW

This error is not normal and should not occur under normal use.

7.2.7 MEMORY FULL

This error is caused when the current program and its associated variables are too large to fit in the memory available for baZic and its programs. If your system contains more memory than which baZic is MEMSET, you can use the MEMSET command to claim more memory. Otherwise the program needs to be broken into smaller "pieces" if possible and linked using the CHAIN statement.

7.2.8 MISSING NEXT

This error is very specific to the FOR NEXT loop. This error means that a FOR NEXT loop was started, but baZic can't find the NEXT statement to define the loop. Every FOR in a program must have a NEXT directly associated with it.

7.2.9 NO PROGRAM

This error only occurs when an attempt is made to use the RUN command but no current program is in the internal memory of the computer.

7.2.10 TOO LARGE OR NO PROGRAM

This error is returned when an attempt is made to LOAD, CHAIN, or APPEND a program which isn't a valid program or the program is too large to fit into the available memory in your system.

This error can occur when a program is partially loaded which contains "garbage" caused by an improperly saved program or from a magnetically damaged disk. A disk can be damaged by turning off the power to the computer without opening the drive door or by opening the drive door while the computer is writing to the disk.

The following internal routine can cause this error to occur:

INTL5 Part of initialization routine which is used on
ORG+4 entry or re-entry following LOAD, CHAIN, or
APPEND. The error occurs on failure to find an end
of program marker. Either the program was too
large for memory or the program is invalid.

MISCELLANEOUS TOPICS

This section is designed to cover several different topics that will be of concern to programmers using baZic. The first section will discuss the internal line editor of baZic. The following sections will discuss how baZic stores information in disk files and the major differences between baZic and other implementations of BASIC.

8.1 Line Editor

To allow the easy creation and changing of programs, baZic has a "built in" line editor. This editor may be used while entering a program, changing a program, or in response to the INPUT or INPUT1 statements.

Internally in baZic are two line buffers. As you type in any line of baZic code, you are typing into the primary input buffer. Once the line has been entered and the carriage return key pressed, the line just typed in is moved to the editor buffer. This means that any line of program just typed is available for editing.

Any other line of text can be placed in the editor buffer by issuing the EDIT command with the desired line number as the argument. The editor works the same no matter which method was used to place a line of text in the editor buffer (i.e. typing in an original line of text or invoking the EDIT command).

To keep track of the changes in a line of baZic program, an internal pointer is used to "point" to characters in both the editor buffer and the input buffer. As every character of text (except the editor commands) is typed in, both pointers are advanced one character at a time so that the pointer to the input buffer is always pointing to the current character position. The editor keeps track of both pointers and allows the programmer to transfer information from the editor buffer to the input buffer.

When the editor is invoked by issuing the EDIT command, baZic automatically displays the line number in question. The editor also "looks" at two different types of backspace characters. If your system uses the underline (ASCII 95), the editor will output an underline to the device to signify the backspace. The editor assumes that you have a teletype-like device that is not capable of backing up. However, if you define your backspace as ASCII 8, the editor upon receiving this code, will issue a backspace (ASCII 8), a space (ASCII 32) and another backspace.

The following commands are available in the line editor and are used to copy characters or sets of characters from the editor buffer to the input buffer: ^G (control G), ^N, ^A, ^Q, ^Z, ^D, and ^Y. All of the editor commands are control characters (the control key is pressed at the same time the character is pressed).

8.1.1 Control G

Control G copies the entire contents of the editor buffer from the current cursor position within the line to the input buffer. Control G may be used to view the editor buffer after a line has been placed in the editor buffer. After the control G has been executed, the programmer should be able to see the line that was in the editor buffer and the cursor will be at the end of the line.

At this point the programmer can take one of two actions: press the return key which enters the edited line into the program, or press the control N command which leaves the line in the editor buffer and returns the cursor to the beginning of the line. This procedure is very useful when viewing a line of text in the editor buffer prior to doing the actual editing.

If there is no line of text or the pointer is already at the end of the text in the editor buffer, the bell will be sounded if a control G command is issued by the programmer.

8.1.2 Control N

The Control N command is partially discussed in the control G section. The purpose of the control N command is to allow the programmer to restart the editing of the line in the editor buffer by cancelling the line presently on the screen and returning the cursor to the beginning of the line for further editing. An "@" sign is printed when the Control N command is typed to indicate to the programmer the line has been cancelled.

8.1.3 Control A

The Control A command is used to copy one character from the editor buffer to the input buffer. The pointers can be pointing to different characters in each buffer so the command "takes" the character pointed to in the editor buffer and places it in the input buffer.

The character is also printed to the CRT as if the programmer had typed the character into the input buffer. Both pointers are incremented after this command. If no character is in the editor buffer, the "bell" is sounded on the CRT to let the programmer know that the command was illegal.

8.1.4 Control Q

This is the backspace command. It is identical to the backspace key on many CRTs or the Control H key. Both pointers are decremented by this command. If your baZic is set to recognize the underline character (ASCII 95) for a backspace, the command prints an underline character each time it is executed to inform the programmer how many characters the command has "backed over."

If your baZic is set to recognize a backspace character (ASCII 8), a backspace is printed followed by a space (ASCII 32) followed by another backspace. If one or both pointers are at the beginning of a line, the command sounds the "bell" of the CRT to let the programmer know of the mistake.

8.1.5 Control Z

This command is used to erase one character at a time from the input buffer. The command prints a "%" sign to inform the programmer that the character position occupied by the "%" sign has been erased and is no longer in the input buffer. If the input buffer pointer is already at the beginning of the line, the bell is sounded to inform the programmer of the error.

8.1.6 Control D

The Control D command is the search and find command. Upon executing the command, baZic will wait for one additional character to be input. Once this character is input, the editor buffer is searched until the first occurrence of the specified character is found and the contents of the editor buffer up to but not including that character is copied to the input buffer. If the character is not located in the editor buffer, nothing is copied to the input buffer and the bell is sounded.

8.1.7 Control Y

The Control Y command is used to "turn on and off" the insert mode. By executing the Control Y command to turn on the insert mode, characters may be inserted into the input buffer that were not in the editor buffer. Once the characters have been entered, Control Y can be toggled off again to allow other characters to be copied or deleted from the input buffer.

When the insert mode is toggled on, a "less than" character (<) will be printed to inform the programmer that the editor is in the insert mode. When the insert mode is toggled off, a "greater than" character (>) is printed to inform the programmer that the insert mode is off. These characters are not part of the line itself but are placed in the line shown on the CRT so the programmer will know the status of the insert mode.

8.2 Data Files

Data files are used for the permanent storage of data collected or ordered by baZic programs. A data file essentially is a section of a disk that is given a file name and a specific location and size. The Disk Operating System (MicroDoZ) is responsible for keeping track of all files, and accessing of these files is done from baZic through the Disk Operating System. See the MicroDoZ manual for more information concerning files and the information that is kept associated with each one.

The file names for data files follow the same rules as all other files under the respective Disk Operating System. The file name is a string value that must be no longer than 8 characters. Any combination of characters is allowed except an ASCII space or a comma. The comma is used to separate the file name from its drive number reference.

Files are limited to approximately 16 million characters (16,776,960 bytes), if your system has this much storage. Otherwise files are limited only by the amount of storage available on your system hardware.

Files under MicroDoZ can have up to 128 different types (0 to 127). Generally baZic data files are given a type of 3 but can be any type if the type argument is specified when the file is OPENed.

Files under MicroDoZ are not dynamic (i.e., they can't expand automatically when the file size is exceeded). This means that all files should be the proper size or greater when created.

Before any file can be used by baZic, it must be OPENed. See Section 3.4.3 for more information on OPENing a file. When the programmer is finished with a file it should be CLOSED as soon as possible so that any data remaining in the internal RAM buffer is "flushed" out of RAM and into the file.

Data files under baZic may contain four types of information; strings, numbers, bytes, and "end marks." End marks are written after every file WRITE that doesn't contain the reserved word NOENDMARK. In sequential writes, the ENDMARK is overwritten by the new record and a new ENDMARK is written after the last data so that normally files contain only one ENDMARK. The inclusion of the NOENDMARK reserved word at the end of any variable list that is to be written to a file will suppress the writing of an end mark. The end mark is represented by a 1 (byte value) in the file.

If strings are written to a file, a variable amount of space is required to store them, depending upon the length of the string. If the string is less than 255 characters, two bytes of "overhead" are required for each string stored. If the string length is over 255 characters, three bytes of overhead are required. This overhead value must be taken into account when reading or writing files in a random manner.

The first byte of a string stored on disk is the byte value of 3 if the string is greater than or equal to 255 bytes or 2 if the string is less than 255 bytes. Strings are written this way so that the TYP function can recognize a string. The next byte (or two bytes if the string is over 255 characters long) is the length of the string.

Numerics take a predefined amount of file storage based upon the precision of the baZic that is writing the file. The precision required to store a numeric in a file is contained in the following table:

PRECISION OF BAZIC	BYTES REQUIRED FOR FILE STORAGE
8	5
10	6
12	7
14	8

Numerics are stored in the files as packed binary-coded-decimal (BCD) values. The digits of the number are stored in the appropriate number of bytes with each digit being stored in a nibble (4 bits). The first byte of a number contains two nibbles, the first nibble being the most significant digit in BCD.

Each nibble from the first byte to the last byte contains each succeeding BCD digit. The number of these bytes is dependent upon the precision. The last byte is the sign and exponent of the number. Bit 7 is the sign (1=negative, 0=positive) and the remaining bits (6-0) are the exponent of the number.

Byte writes, of course, only take one byte of storage. However, a NOENDMARK is generally required when writing bytes so that an end mark is not written after every byte.

When reading and writing strings and numerics, baZic uses a very structured approach so that it can always recognize these two entities. The TYP function can be used to determine the type of data that is currently at the file pointer location. The TYP function can recognize strings (type 1), numbers (type 2), and end marks (type 0).

When a file is OPENed, an internal buffer is established in baZic for reading and writing files. This buffer is 512 bytes long for each channel opened. When a file is closed, the buffer immediately is written back to the disk (but not in the case where only read operations were performed on the file). The memory space occupied by the buffer is not recovered but if the channel is opened again the same buffer will be used.

8.3 BASIC Differences

Every BASIC on the market is different from every other BASIC in some way. baZic has been written to be quite similar to North Star BASIC in operation but is different in several ways. North Star BASIC is different from most other BASICs on the market, mainly in its string handling capabilities.

8.3.1 Strings

Most BASICs allow strings to be only 255 characters long but support string arrays. baZic allows strings to be any length, limited only by internal memory, but string arrays are not supported per se. String arrays can be simulated very easily under baZic.

A string array can be simulated by DIMensioning a string large enough to hold the entire array. A simple string position calculation is then performed to access the "element" of the array that is needed. The following example shows how to access the Eth element (E is the element number) in string A\$, where each element is L bytes long:

```
A$(E*L-(L-1),E*L)
```

All strings greater than 10 bytes long must be DIMensioned before they are used. If a string is used in a program, it is automatically DIMensioned to a length of 10. Strings can be any length limited only by available memory.

baZic does not use LEFT\$, MID\$, and RIGHT\$ as many other BASICs do. baZic has a much more convenient method of defining substrings within a larger string. The position of the substring is passed to any string argument exactly the same way as the string name would be. The following examples show the method of converting LEFT\$, MID\$, and RIGHT\$ to equivalent baZic string representations:

LEFT\$(A\$,L)	would be	A\$(1,L)
RIGHT\$(A\$,R)	would be	A\$(LEN(A\$)-R+1)
MID\$(A\$,L,R)	would be	A\$(L,L+R-1)

A\$(L) is the string of characters from position L to the end of the string.

8.3.2 IF THEN Evaluation

Many BASICs evaluate the IF THEN condition differently than baZic. In baZic, if the IF THEN evaluation is false, baZic skips over one statement following the THEN statement and executes the next instruction in that line or the next line if there are no other statements on the IF THEN line. Many other BASICs skip everything else on the IF THEN line if the evaluation of the IF THEN clause is false.

The statement separator of baZic acts exactly as an implied ELSE statement when the initial IF THEN evaluates false. The following example shows how baZic handles the IF THEN statement:

```

10 X=0
20 Y=6
30 GOSUB 70\REM PRINT VALUE OF X AND Y
40 IF X=Y THEN 50\GOSUB 70\Y=21\X=21\GOTO 30
50 GOSUB 70\REM PRINT VALUE OF X AND Y
60 END
70 PRINT "X=",X\REM SUBROUTINE TO PRINT VALUE OF X AND Y
80 PRINT "Y=",Y
90 RETURN

```

Upon RUNNING the preceding program, the following would appear on the CRT:

```

READY
RUN
X= 0
Y= 6
X= 0
Y= 6
X= 21
Y= 21
X= 21
Y= 21

```

8.3.3 Miscellaneous

Many other BASICs return a -1 if the result of a relational operation is true. baZic returns a +1. In both cases, a zero (0) is returned if the operation is false.

Many BASICs do not support the zero dimension of an numeric array. When an array is dimensioned to 10 under baZic (B(10)), the array actually has 11 elements (0 to 10). If memory is at a premium in your system and you only need 10 elements, dimension your arrays to 9 (B(9)).

The NEXT control variable is incremented in baZic upon finishing a FOR NEXT loop. Many BASICs do not increment this variable at this time. Upon leaving a FOR NEXT loop, the control variable will be one step value greater than the limit value.

UTILITY PROGRAMS

The following are assembly language programs designed to run in conjunction with baZic. SHORTB is a program that removes the mathematical functions from baZic to regain more usable RAM work space. XREF does a variable cross reference of a baZic program that is loaded into RAM.

9.1 SHORTB

The SHORTB program is not available with the CP/M version of baZic.

For many applications the baZic mathematical routines, SIN, COS, ATN, etc. are excess baggage. The assembly language program, SHORTB, can be used to delete unused routines from baZic. The deletion is irreversible. **KEEP A SAFE COPY of the original.**

The following instructions for using SHORTB assume that you are in baZic.

```
Type    BYE
Type    GO SHORTB,n    where n is the applicable
                        drive #.
```

A selection menu will appear:

KEY THE APPROPRIATE NUMBER TO REMOVE FUNCTIONS FROM ATN
THROUGH ...

1. ATN
2. SIN-COS
3. LOG
4. EXP
5. RAISE TO POWER, ^
6. SQRT
7. RND
8. NONE OF THE ABOVE

The routine essentially moves the end of baZic. It is not possible to delete SQRT, for example, and retain ATN, SIN-COS, etc. Selection of option 6 deletes ATN, SIN-COS, LOG, EXP, ^, and SQRT. In addition to moving the end of baZic; SHORTB substitutes a JMP to SYNTAX ERROR for all internal calls to the affected routines. This avoids visits to never-never land if you should attempt to run a program which uses a deleted routine.

The RAISE TO POWER operator (or ^ as it appears in baZic programs) requires additional explanation. baZic uses the LOG and EXP routines to evaluate ^ for non-integer powers, negative powers, and all powers greater than 30. Hence, if you delete LOG, the expression, 12^3 , will be evaluated properly. However, the expressions, $12^{3.1}$, 12^{-3} , and 12^{31} will result in SYNTAX ERRORS.

On completion of the selected modification SHORTB does a JMP to the 2D00H entry point of baZic. Selection of item 8, NONE OF THE ABOVE, returns you to baZic without modifying baZic. In all cases, any program which was in RAM will be lost.

If you want a permanent copy of the shortened baZic, you must exit to DOS by typing BYE and save baZic on the disk by typing SF BAZIC 2D00 (DOS version). The procedure would appear as follows:

```

READY
BYE
+SF BAZIC 2D00
+JP 2D00
READY

```

Table 1 lists the number of bytes of memory that can be recovered for your programs or data by selecting the various options. The savings are the same for both the hardware floating point and software floating point versions.

Table 1

BYTES RECOVERED BY DELETING MATHEMATICAL FUNCTIONS

	baZic08	baZic10	baZic12	baZic14
ATN	166	177	188	199
COS-SIN	395	432	454	476
LOG	657	704	750	784
EXP	957	1015	1079	1125
^	1025	1083	1147	1193
SQRT	1174	1234	1300	1348
RND	1266	1326	1392	1440

9.2 XREF

The XREF program is not available in the CP/M version of baZic.

XREF is an assembly language program which analyzes a baZic program and prepares a table of the variables used and the program lines in which these variables occur. Such a table can be extremely useful in documenting or de-bugging your programs. XREF requires RAM from BD00 to BFFF Hex to operate.

The following instructions for using XREF assume that you are in baZic and have your program loaded:

Type BYE
Type GO XREF,n where n is the applicable drive #.

When the program is loaded and executed the prompt message will appear:

Select print option:

- 1) CRT
- 2) Printer

Printing of the table begins immediately on selection. Table 2 is an XREF table for the baZic program listed under Listing 1. The number of symbols used by the program and the total number of times all symbols are used are also printed. On successful completion of the listing, or if an error is encountered, XREF returns to the 2D04H entry point of baZic, leaving the baZic program undisturbed.

XREF makes no attempt to determine if the baZic program will run. However, there are three programming errors which will cause XREF to abort. These are:

1. Quotation marks not closed.
2. No comma after a print format specification.
3. No letter after FN.

In each case the error message will be the same:

ERROR IN LINE xxx

If you examine the indicated line and do not find one of the three errors listed above, there is a fourth possibility unique to XREF. As currently written, XREF cannot list a program in which the same symbol is referenced in more than 126 program lines. The probability of this occurring is small.

The fifth and final error condition which may be encountered in using XREF is a MEMORY FULL error. XREF occupies memory area from BD00H to BFFFH. Data for the output is accumulated between the end of your baZic program and the beginning of XREF. For a very long program memory capacity may be exceeded.

It is not necessary to perform a MEMSET prior to running XREF. It in no way affects the baZic program, unless of course your program extends to or beyond BD00H. The return to baZic will however, wipe out the copy of XREF in RAM. If you want a second copy of the variable table you must again execute GO XREF.

This program will work with all eight versions of baZic, i.e., 8, 10, 12, and 14 digit precision in both the hardware floating point and software floating point versions.

Listing 1

```

10 REM A program to determine BASIC memory requirements.
20 REM Filename MEMREQR
30 O=11520\ REM Address of first byte of BASIC.
40 INPUT"BASIC USED ",A$
50 INPUT"Key 1 for printer, 0 for CRT, 2 for both.",P
60 IF A$(3,3)="Z" THEN X=31 ELSE X=25
70 E1=256*EXAM(O+7)+EXAM(O+6)\REM Address of last byte of BASIC.
80 A=EXAM(O+X)+256*EXAM(O+X+1)\REM Address of ATN routine
90 C+EXAM(O+X+2)+256*EXAM(O+X+3)\REM Address of COS routine
100 L=EXAM(O+X+4)+256*EXAM(O+X+5)\REM Address of LOG routine
110 E=EXAM(O+X+6)+256*EXAM(O+X+7)\REM Address of EXP routine
120 IF A$(3,3)="S" THEN 160
130 T=EXAM(O+X+8)+256*EXAM(O+X+9)\REM Address of ^ routine
140 S=EXAM(O+X+10)+256*EXAM(O+X+11)\REM Address of SQRT routine
150 R=EXAM(O+X+12)+256*EXAM(O+X+13)\REM Address of RND routine
160 !#P,"Memory requirements for ",A$
170 !#P,"ENDBAS at",E1,TAB(20),"Total memory occupied is",E1-0+1,
    " bytes."
180 !#P,"ATN AT",A,TAB(20),"Deletion recovers",E1-A," bytes."
190 !#P,"COS at",C,TAB(20),"Deletion recovers an additional",A-C,
    " bytes."
200 !#P,"LOG at",L,TAB(20),"Deletion recovers an additional",C-L,
    " bytes."
210 !#P,"EXP at",E,TAB(20),"Deletion recovers an additional",L-E,
    " bytes."
220 IF A$(3,3)="Z" THEN 250
230 !#P,"Total memory recovered by deleting all four routines is",
    E1-E," bytes."
240 GOTO 290
250 !#P,"^ at",T,TAB(20),"Deletion recovers an additional",E-T,
    " bytes."
260 !#P,"SQRT at",S,TAB(20),"Deletion recovers an additional",T-S,
    " bytes."
270 !#P,"RND at",R,TAB(20),"Deletion recovers an additional",S-R,
    " bytes."
280 !#P,"Total memory recovered by deleting all seven routines
    is",E1-R," bytes."
290 !#P\!#P
300 END

```

Table 6

A	80,180,190
A\$	40,60,120,160,220
C	90,190,200
E	110,210,230,250
El	70,170,180,230,280
L	100,200,210
O	30,70,80,90,100,110,130,140,150,170
P	50,160,170,180,190,200,210,230,250,260,270,280,290
R	150,270,280
S	140,260,270
T	130,250,260
X	60,80,90,100,110,130,140,150

Symbols 12
References 88

9.3 ICOPY

This program copies in A)uto or S)ingle step mode. A)uto mode copies file-by-file until the entire disk is copied or will stop when the destination filespace is too small to continue or a duplicate filename is encountered. S)ingle step mode copies file-by-file but prompts the operator as to the action to be taken for each file.

The first prompt is the request for the drive number to copy from:

```
INTELLIGENT COPY COPYRIGHT (C) 1981, MICRO MIKE'S, INC.
INPUT SOURCE DRIVE NUMBER
#
```

followed by the request for the drive to copy to:

```
INPUT DESTINATION DRIVE NUMBER
#
```

If the user enters the same response to both drive number requests, the program will print the following message and return to the first prompt:

```
SOURCE DRIVE CAN'T EQUAL DESTINATION DRIVE
```

The mode is selected in response to the third prompt:

```
CHOOSE A)UTO OR S)INGLE STEP
*
```

To select your choice respond with the first letter of the name of the option you want ("A" for auto or "S" for single step). The A)UTO option begins copying from the point where it is invoked to the end of the disk. The S)INGLE STEP option allows the user to view each file before deciding if the file is to be copied.

If the A)UTO mode is selected, the screen will appear as follows:

```
INTELLIGENT COPY COPYRIGHT (C) 1981, MICRO MIKE'S, INC.
```

```
C
```

```
FILENAME          AF=0
```

The file name and the attribute will display on the fourth line of the CRT as the files are being copied. When the copy is complete, the program will respond with the following prompt:

```
INPUT E)XIT OR C)ONTINUE
```

If the user selects the "E" option the user is returned to the DISKETTE UTILITY MENU. If the user selects the "C" option, the program will branch to the initial prompt (INPUT THE SOURCE DRIVE NUMBER).

Several conditions will terminate the A)UTO mode. If space on the destination disk (the one being copied TO) is too small the program will terminate with the message:

```
DISK IS FULL
```

If the directory space becomes full, the program will issue the following message:

```
DIRECTORY IS FULL
```

If a hard disk error occurs, the program will display the following message:

```
HARD DISK ERROR ON DRIVE X
```

If the destination disk is write protected (write protect tab in place), the program will respond with the following message:

```
FILE IS WRITE PROTECTED
```

The occurrence of any of the these messages will result in the following prompt:

```
INPUT E)XIT OR C)ONTINUE
```

If the user selects the "E" option the user is returned to the DISKETTE UTILITY MENU. If the user selects the "C" option, the program will branch to the initial prompt (INPUT THE SOURCE DRIVE NUMBER).

If the program encounters a file that already exists on the destination disk the program will respond with the following prompt and message:

```

CHOOSE S)KIP, R)EPLACE, N)EWMNAME, OR E)XIT
*
DUPLICATE FILE NAME

```

The S)KIP option skips the present file and goes to the next file. The file is NOT copied if you specify S)KIP.

The R)EPLACE option will cause the file on the destination disk to be over-written by the file of the same name on the source disk (the disk being copied from). The program will then continue to the end or will stop if any of the terminating conditions are found again (file space too small or duplicate file name).

The N)EWMNAME option will allow you to give the file a new name. In this instance you will still retain the file on the destination drive with the duplicate name but the file will be copied from the source to the destination drive and be named whatever you choose.

If the N)EWMNAME option is chosen, the program will ask for the new file name with the following prompt:

```

INPUT NEW FILE NAME
*****

```

If the name is less than eight characters, enter a RETURN after the name. If the name is eight characters the program will supply the RETURN.

The E)XIT option allows you to exit to the error condition prompt:

```

INPUT E)XIT OR C)ONTINUE

```

If the user selects the "E" option the user is returned to the DISKETTE UTILITY MENU. If the user selects the "C" option, the program will branch to the initial prompt (INPUT THE SOURCE DRIVE NUMBER).

In S)INGLE STEP mode the program works similarly to the auto mode except each file is displayed on the CRT one at a time, allowing the operator to make decisions based on the file name. Under S)INGLE STEP the program will display the name of the first file on the source drive and issue the following prompt:

```

CHOOSE C)OPY, S)KIP, A)UTO, OR E)XIT
*
FILENAME          AF= 0

```

The C)OPY option performs a copy of the file and then displays the next file. If a duplicate record is encountered, the program performs exactly as stated in the A)UTO mode description of duplicate file names.

The S)KIP option skips the currently displayed file and brings up the next file for the operator to act upon.

The A)UTO option allows the user to A)UTO copy all files on the disk. A)UTO may be invoked anytime under S)ingle Step to copy to the end of the disk. The previous description of A)UTO applies anytime it is invoked.

The E)XIT option works exactly as previously described.

All files are compacted automatically as they are copied. Using ICOPY, files are transferred onto the "tail end" of any data already on the disk being copied to. Filenames are displayed on the CRT screen as they are being copied. ICOPY employs read-after-write verification of data being transferred.

Any time the user is at the INPUT E)XIT OR C)ONTINUE prompt and the user specifies the "E" option, the program will examine the default drive to determine if the proper programs are on that disk. If the user makes a copy of a disk and forgets to place the utility disk back in the default drive, the correct programs will not be on the disk in the default drive.

When this situation occurs, the program will respond with the following prompt and will wait for you to place the correct disk in the default drive and press return to continue:

UTILITY DISK NOT IN DRIVE X PRESS RETURN TO CONTINUE

9.4 COPYFILE

The copy file program allows the user to copy a single file or multiple files from the user's choice of drives. This program will not normally be used by the end user due to the complexity of the program. This program is "programable" in that all the questions asked by this program can be answered by another program. In this mode, the program will automatically answer its own prompts and no operator intervention is required.

The program first prompts the user for the drive number containing the file(s) to be copied. The prompt appears as follows:

COPY FILE COPYRIGHT (C) 1981, MICRO MIKE'S, INC.
INPUT SOURCE DRIVE NUMBER
#

After entering the source drive number the program asks for the drive number to copy the file(s) to:

INPUT DESTINATION DRIVE NUMBER

#

If the source drive number is equal to the destination drive number, the program will consider the file copy a special case. This situation will be discussed later. The "normal" flow of the program is for the source and destination drives to be different. For this situation (drives different), the program begins a series of prompts requesting information on how you want the file copies to be made. The first prompt in this series is:

DO YOU WANT TO RENAME FILES? Y OR N

Often the file you want to copy from is already created on the destination disk. The previous prompt is asking if you want to automatically rename each file that is encountered on the destination disk that has the same name as the file being copied from the source disk. Answer "Y" if you want all duplicate files automatically renamed or "N" if you do not want duplicate files renamed.

The next prompt is:

DO YOU WANT ALL DUPLICATE FILES REPLACED? Y OR N

If you answer "Y" for this prompt, any duplicate files on the destination disk will be replaced automatically by the file of the same name on the source disk. A "N" response will allow you to decide individually what action is taken for each duplicate file name encountered.

The next prompt is:

DO YOU WANT TO USE A FILE OF FILE NAMES? Y OR N

The "N" option specifies that the user will enter the names of each file to be copied. Alternately, the user may have a file established that contains the names of all the files that are to be copied. Selecting the "Y" option places the file copy program in a mode whereby the program can "look" at the specified file and perform a copy of all files listed in the file of file names. If the "Y" option is selected the next prompt will be:

INPUT FILE NAME

The file name of the file of file names should be entered. If the file is not 8 characters long, a carriage return will have to be entered by the user otherwise the program will supply the carriage return.

The next prompt reads as follows:

DO YOU WANT TO SET INDIVIDUAL FILE DENSITIES? Y OR N

This option allows the user to set the densities (single or double) for each file encountered in the current copy session. If the "Y" option is selected, the user will be prompted for the file density for each file copied. If the "Y" option is selected, the following prompt will be displayed for each file to be copied:

```
INPUT DESTINATION FILE DENSITY.  S OR D
```

The user may specify the density for each file copied. If the "N" option is selected to the DENSITY prompt, the program will automatically set the density of each file copied to the density of the directory on the destination disk. The density of the destination disk is determined when the disk is initialized. Normally only double density is used.

The next prompt to be displayed is:

```
ALL FILE ATTRIBUTES=0 DO YOU WANT TO SET THE FILE ATTRIBUTES?
*
```

Each file on the disk can have one of 64 attributes (0 to 63). This prompt is asking if you want to be able to select the attributes of each file that is to be copied. If you select "N" for no, the program will copy each file to the destination disk and preserve the attribute of the file so that the destination file attribute is the same as the source file attribute. If you select the "Y" option, the program will respond with the following two prompts for each file to be copied:

```
INPUT THE SOURCE FILE ATTRIBUTE.  0..63
##
```

```
INPUT THE DESTINATION FILE ATTRIBUTE.  0..63
##
```

At this point, the program is ready to take in the file name to be copied. The prompt appears as follows:

```
INPUT FILE NAME TO COPY
*****
```

Enter the name of the file to be copied. If the file name is not 8 characters long, enter the carriage return after the file name is entered. If the file name is 8 characters in length, the program will supply the carriage return.

If the file name selected is not on the destination drive, the program will create a file the same size as the file on the source disk and will copy the file to the destination disk. If the file already exists on the destination disk, the program will check the responses to the previous questions to determine what action it is to take.

If the answer to the auto replace prompt was positive, the file on the destination disk will be automatically replaced by the file of the same name on the source disk. The program will make the replacement and print the following message on the CRT to inform the user the auto replace mode is in effect:

AUTO REPLACE

If the new name option was selected as positive, the program will automatically prompt the user for the new name the file is to be given when it is copied to the destination disk. The prompt appears as follows:

INPUT NEW FILE NAME

Once the copy is completed, the program will ask the user if he wants to exit the program, copy another file using the answers to all previous questions, or start over with the program and answer all questions again. The prompt appears as follows:

CHOOSE E)XIT, M)ORE FILES, OR R)ESTART COPY

The "E" option always takes the user to the program that originally called the COPYFILE program. The "M" option allows the user to copy more files. Under this mode, the user is not asked for the source and destination drive number, or the other questions pertaining to how the copy is to be made. Selecting the "R" option takes the user back to the original prompt of the program (INPUT SOURCE DRIVE NUMBER).

If, at anytime during the copy, the user encounters a duplicate file and the user specified that the program was NOT to automatically replace the destination file with the source file, the program will display the following prompt and message:

CHOOSE E)XIT, S)KIP, R)EPLACE, OR N)EWSNAME
*
DUPLICATE FILE NAME

If the user selects the "E" option the program displays the following prompt:

INPUT E)XIT OR C)ONTINUE

The "E" option in this case takes the user back to the calling program. The "C" option allows the user to start over at the original prompt (INPUT SOURCE DRIVE NUMBER).

If the user selects the "S" option the file is skipped and is not copied from the source disk to the destination disk.

The "R" option specifies that the destination file is to be overwritten by the source file.

The "N" option allows the file to be copied from the source disk to the destination disk and given a new name on the destination disk. The program issues the following prompt when the "N" option is selected:

INPUT NEW FILE NAME

If the initial two prompts (INPUT SOURCE AND DESTINATION DRIVE NUMBERS) result in the same drive number being input by the user, the program will continue to function but two prompts will not be issued as in the normal sequence. These two prompts are:

DO YOU WANT TO RENAME FILES? Y OR N

DO YOU WANT TO ALL DUPLICATE FILES REPLACED? Y OR N

The program assumes that duplicate files will be encountered since the source disk and destination disk are the same. The general procedure in this situation is to rename the file as described previously. The DUPLICATE FILE NAME messages and prompts will also be as previously described.

During the course of a file copy, several error messages may be displayed on the CRT. The following messages are essentially self explanatory:

FILE NOT FOUND
 DESTINATION DIRECTORY IS FULL
 DESTINATION DISK IS FULL
 DUPLICATE FILE NAME

In the automatic mode where the program is using a file of files, there are two additional messages that are displayed on the CRT simply to inform the user of proper operation of the program:

READING FILE NAMES FROM FILE
 END OF FILE NAME FILE

9.5 COMPACT

The compact program is used to recover file space that has been lost on a drive due to the deletion of files that were not at the end of the data area. The compact option re-organizes the data on a disk so that all the files are at the beginning of the disk and all of the "blank" room is at the "end" of the disk. **A copy should always be made of a disk before it is compacted.**

The first prompt of this option is:

COMPACT COPYRIGHT (C) 1981, MICRO MIKE'S, INC.
 INPUT DRIVE TO COMPACT
 #

If a hard disk error occurs during the compaction, the program will print the following message:

HARD DISK ERROR ON DRIVE X

When the compaction is complete or an error occurs, the following prompt is displayed:

INPUT E)XIT OR C)ONTINUE

If the user selects the "C" option to continue, the program branches to the first prompt of the compaction (INPUT DRIVE TO COMPACT). The "E" option takes the user to the DISKETTE UTILITY MENU.

Any time the user is at the INPUT E)XIT OR C)ONTINUE prompt and the user specifies the "E" option, the program will examine the default drive to determine if the proper programs are on that disk. If the user has made a copy of a disk and forgot to place the utility disk back in the default drive, the correct programs will not be on the disk in the default drive.

When this situation occurs, the program will respond with the following prompt and will wait for you to place the correct disk in the default drive and press return to continue:

9.6 Interfacing COPY Programs to baZic Programs

All of the initialize, copy, and compact programs can be interfaced very easily to MicroDoZbaZic. All functions are used by either CHAINing to the appropriate program or by using the DOSCMD feature of baZic to call a machine language program (ICOPY, COPYFILE, COMPACT).

ICOPY, FILECOPY, and COMPACT can be used by a baZic program. In the case of these three programs, the program is called by using the DOSCMD statement of baZic. The DOSCMD statement can be used to execute a GO ICOPY command from baZic.

When a DOSCMD statement is used, the contents of the entire command being passed to MicroDoZ is moved to the CCB (Common Command Buffer) located at 80H. MicroDoZ then acts on the command in the CCB.

The programs ICOPY, FILECOPY, and COMPACT all have the additional capability of using values in the CCB to answer the prompts displayed when the program is executed. When any of these programs are executed, they locate the next separator byte in the CCB. If two separators are together, the programs assume that no commands follow and they use the file name after the second separator as the program to load when the copy program is exited.

If the copy programs do not find a double separator at the current separator byte, the programs assume that what follows in the buffer is additional commands which are to be used by the program to answer its own prompts.

As an example, the following sequence could be sent to MicroDoZ from baZic and would result in the files listed in the file FNAME being copied from Drive 1 to Drive 2 without operator intervention. When FILECOPY has finished executing the copy, the program will return to baZic which will LOAD and execute the program "CSUB." The MicroDoZ command appears as follows:

```
10 DOSCMD "COPYFILE\1\2\N\Y\Y\FNAME \1\N\N\E\BAZIC\CSUB"
```

The result of this command sequence would be to execute the program named COPYFILE. Once COPYFILE is executed it will respond with the following prompts which are answered by the values in the CCB. Each response that is in the CCB will be underlined in the series of questions that follow. The sequence of prompts is as follows:

```
COPY FROM DRIVE 1
COPY TO DRIVE 2
AUTO RENAME N
AUTO REPLACE IF DUPLICATE FILE NAME Y
USE A FILE OF FILE NAMES Y
NAME OF FILE NAME TO USE FNAME
FILE IS ON DRIVE 1
PROMPT DENSITY FOR EACH FILE N
PROMPT ATTRIBUTES FOR EACH FILE N
INPUT E)XIT OR C)ONTINUE E
```

(The file-of-file-names file should be written sequentially, with each file name stored as a string of 8 characters or less. The list of file names should end with an endmark.)

This versatility of MicroDoZbaZic allows the programmer great power in programming. No longer does the end user have to know the Disk Operating System, just to initialize a disk or make a backup copy. All these things can be set up by the programmer and the end user has only to press a key on the keyboard to accomplish these tasks.

9.7 CRT

CRT is the program used to configure the CP/M and DOS versions of baZic for cursor addressing and clear screen for use with different CRTs. CRT is a baZic program and is LOAded and RUN in the normal manner. Before CRT can be used to change the disk version of CP/M baZic, the version to be changed must be RENamed to a file BAZIC10.COM to BAZIC10.001:

```
A>REN BAZIC10.001=BAZIC10.COM
```

Once the baZic has been RENamed and configured for your CRT, it must be RENamed back to a .COM file to be executed. Since CP/M does not allow a .001 file to be executed, a different version of baZic must be used to convert the version you want changed to your CRT. Any precision of baZic can be used to configure any other precision.

The program will display a menu of CRTs and if your CRT is one of those listed then select the appropriate number and baZic on the disk will be modified. The first menu will be similar to the following:

```
1=ZENITH Z-19 OR HEATH WH-19
2=ADM-3A
.
.
6=OTHER
```

Because the copy of baZic on the disk will be modified, please use a copy of the baZic disk and not the original. The disk copy of baZic is modified by using the BYTE write feature of baZic to modify itself. The location of the bytes to be modified is shown in the partial source listing provided. This modification can also be made from a monitor if a machine language programmer is available.

If your CRT is not listed on the menu and the cursor addressing and clear screen are simple sequences of two characters or less, the program CRT will allow you to enter this information and the program will write to the baZic file as well as the copy of baZic presently in the RAM of your computer. If your cursor addressing and clear screen sequences are not short or uncomplicated, baZic as delivered may not be capable of these functions on your CRT. Special assemblies are available from Micro Mike's, Inc. for additional fees.

After selecting your CRT from the first menu the program will respond by displaying a second menu with a choice of which baZic to modify. The listing includes all versions of baZic that are provided. The second menu will appear as follows:

```
1= 8 DIGIT SOFTWARE FLOATING POINT BAZIC
2= 10 DIGIT SOFTWARE FLOATING POINT BAZIC
3= 12 DIGIT SOFTWARE FLOATING POINT BAZIC
4= 14 DIGIT SOFTWARE FLOATING POINT BAZIC
5= 8 DIGIT HARDWARE FLOATING POINT BAZIC
6= 10 DIGIT HARDWARE FLOATING POINT BAZIC
7= 12 DIGIT HARDWARE FLOATING POINT BAZIC
8= 14 DIGIT HARDWARE FLOATING POINT BAZIC
```

At this point the program will branch one of two ways depending upon which option you selected from the first menu. If you selected one of the CRTs displayed, the program will immediately change the copy of the specified baZic on the disk and the copy that is presently in the computer in RAM. If you select the last option (OTHER), the program will ask you additional information.

Selecting the "OTHER" option will result in the program asking you for the cursor addressing prefix, the offset values for the row and column positions, and the clear screen code(s). Instructions are provided in the program and they should be followed closely. After all the information is entered, the program will branch back to the testing section to see if the changes made work properly.

After the changes to baZic are completed a testing phase is initiated. The first test is the clear screen (CLS) statement. If this statement is working properly you should see the screen clear and a message printed at the top of the CRT. If everything appears in order at this stage, respond to the prompt by entering a "1." Entering a "0" will cause the program to branch back to the beginning menu.

If you entered a "1" indicating that the clear screen worked the next test will be the cursor addressing test. The screen will again be cleared and this time a distinct pattern will be printed on the CRT. The pattern will appear similar to a "rope" looped around itself several times (i.e. a prolate cycloid).

If this pattern appears, you are through using this program and can exit by answering the last prompt with a "1" indicating that the clear screen and cursor addressing have been installed properly for your CRT. A no or "0" answer to the final prompt will take you back to the beginning menu to let you try again.

Remember that running this program changes only the precision of baZic you have requested. If you want another baZic changed you will have to run the program again for each version of baZic you want changed.

9.8 Other CRTs

If your CRT is not listed in the menu of the IOEDIT or CRT program, you will be required to manually "patch" the CRT information before the PRINT@ and CLS statements will function correctly. Consult the partial source listing provided in Section 11 to find the positions within baZic (or MicroDoZ) where the proper codes should be placed. Also see the JOEMASTER PROGRAMMER manual for more information of configuring a CRT for MicroDoZ.

OPTIMIZING baZic PROGRAMS

Programs written under baZic can be optimized to run faster by following the general rules as set forth in this section.

In general, subroutines will execute faster than a similar routine that is written as a user-defined function. However, there are still advantages to user-defined functions that may outweigh the speed advantage of a subroutine.

Since baZic (as well as BASIC) uses a branching linked structure to "find" variables, programs will execute faster if different variable names are used instead of different variables with the same first letter such as F1, F2, F3 etc.

The reason for this is that the location of each letter is known to the interpreter but for each variable with the same first letter the interpreter must search through all the variables with that letter name to find the one in question.

If you do use the same variable letter, use or DIMension the most used early in the program. Variables are assigned space on a "first-come, first-served" basis, so those defined or used early get the "closest" space.

To increase the execution speed of a baZic program, frequently executed subroutines should be at the beginning of the program, especially if there are branches to other line numbers.

PARTIAL SOURCE LISTING

This section is designed to allow the user to change easily the features of baZic that can be changed by the user. A partial source is included for each version of baZic.

11.1 MicroDoZbaZic

```

*
0100      ORG ORIGIN
0100 AF    ENTRY1 XRA A      Clears program
0101 C3 0105  JMP RENTR1
0104 37    ENTRY2 STC       Saves program
0105 21 3230 RENTR1 LXI H,ENDBAS
0108 11 8000  LXI D,08000H   End memory
010B C3 0133  JMP INTL1
***
010E 50    LINECT DB 50H    Initial line length
010F 01    AUTOS DB 1      Auto-start flag
0110 E8    PROM DB 0E8H    Disk controller origin
0111 27A8  LINETB DW PHPOS  Print head table
0113 18    PAGES DB 18H    CRT Lines per page
*
0114 C3 0201  ENTRY3 JMP READY1  Saves data
*
0117 08    BSPC DB 8H      Character delete
0118 00    CONC DB 0       Control C inhibit flag
0119 0007  DS 7
0120 0003  DS 3
;
0123 317F  SBADATA DW ATN    Addresses used by SHORTB
0125 3080  DW COS          program.
0127 2F70  DW LOG
0129 2E39  DW EXP
012B 2DF5  DW TOPWR
012D 2D5E  DW SQRT
012F 2D12  DW RND1
0131 2E37  DW TOPWR4+1

```

Control C can be inhibited by FILLing the byte at 280 Decimal with a byte value of 1. Control C can be re-enabled by FILLing this byte back to zero (0).

baZic can be made to automatically RUN a program upon boot-up by setting the AUTOS byte (010FH) to 0 (FILL 271,0) and saving baZic with a program attached. A file should be made first that is large enough to contain baZic and the turnkey program. Boot-up baZic and LOAD or write the turnkey program.

FILL the turnkey byte with the proper value (0) and BYE to the Disk Operating System. Save baZic with its turnkey program as a single file from its normal operating location by using the SF (Save File) command of DOS.

Alternately, MicroDoZ can be used to "send" a turnkey command to baZic.

11.2 baZic for CP/M

CPM BAZIC

```

    0100          ORG ORIGIN
0100 AF          ENTRY1 XRA A           Clears program
0101 C3 0105     JMP RENTR1
0104 37          ENTRY2 STC           Saves program
0105 21 3197     RENTR1 LXI H,ENDBAS
0108 11 8000     LXI D,08000H         End memory
010B C3 0165     JMP INTL1
***
010E 50          LINECT DB 50H        Initial line length
010F 01          AUTOS DB 1           Auto-start flag
0110 E8          PROM DB 0E8H        Disk controller origin
0111 28B5       LINETB DW PHPOS       Print head table
0113 18          PAGES DB 18H        CRT Lines per page
*
0114 C3 0253     ENTRY3 JMP READY1    Saves data
*
0117 08          BSPC DB 8H           Character delete
0118 00          CONC DB 0            Control C inhibit flag
;           Cursor addressing routine. The example below i
;           for the Zenith WH19 CRT. On entry register A
;           contains the device number, register D the
;           column and register E the line.
FFFF          IF CPM
0119 D5          GOTOYX PUSH D        Zenith WH19 cursor addressing
011A 0E 06       MVI C,6              escape sequence is ESC Y
011C 1E 1B       MVI E,27
011E CD 0005     CALL BDOS
0121 0E 06       MVI C,6
0123 1E 59       MVI E,"Y"
0125 CD 0005     CALL BDOS
0128 21 1F1F     LXI H,1F1FH         Zenith WH19 offset
012B D1          POP D
012C 19          DAD D
012D E5          PUSH H
012E 0E 06       MVI C,6
0130 5D          MOV E,L              The line
0131 CD 0005     CALL BDOS
0134 E1          POP H
0135 0E 06       MVI C,6
0137 5C          MOV E,H              The column
0138 CD 0005     CALL BDOS
013B C9          RET
ENDIF

```

```

013C 0007          DS 7

;          Clear screen routine. The example below
;          is for the Zenith WH19 CRT. On entry
;          register A contains the device number.
;
          FFFF          IF CPM
0143 0E 06          CLS1 MVI C,6          Direct console output
0145 1E 1B          MVI E,27          Zenith WH19 clear screen
0147 CD 0005          CALL BDOS          sequence is ESC "E"
014A 0E 06          MVI C,6
014C 1E 45          MVI E,"E"
014E CD 0005          CALL BDOS
0151 C9          RET
          ENDIF
0152 0003          DS 3

;
0155 30F1          SBDATA DW ATN          Addresses used by SHORTB
0157 300C          DW COS          program.
0159 2F06          DW LOG
015B 2DDA          DW EXP
015D 2D96          DW TOPWR
015F 2D01          DW SQRT
0161 2CB5          DW RND1
0163 2DD8          DW TOPWR4+1

```

*

Control C can be inhibited by FILLing the byte at 0118H with a byte value of 1. Control C can be re-enabled by FILLing this byte back to zero (0).

baZic can be made to automatically RUN a program upon boot-up by setting the AUTOS byte (010FH) to 0 (FILL 271,0) and saving baZic with a program attached. Boot-up baZic and LOAD or write the turnkey program. FILL the turnkey byte with the proper value (0) and BYE to the Disk Operating System. Save baZic with its turnkey program as a single file from its normal operating location by using the SAVE command of CP/M.

Alternately, baZic can be turnkeyed from CP/M by following the name of the baZic by the program to be executed.

ASCII TABLE

ASCII	DECIMAL	HEX	OCTAL	ASCII	DECIMAL	HEX	OCTAL
NUL	0	00	000	SP	32	20	040
SOH	1	01	001	!	33	21	041
STX	2	02	002	"	34	22	042
ETX	3	03	003	#	35	23	043
EOT	4	04	004	\$	36	24	044
ENQ	5	05	005	%	37	25	045
ACK	6	06	006	&	38	26	046
BEL	7	07	007	'	39	27	047
BS	8	08	010	(40	28	050
HT	9	09	011)	41	29	051
LF	10	0A	012	*	42	2A	052
VT	11	0B	013	+	43	2B	053
FF	12	0C	014	,	44	2C	054
CR	13	0D	015	-	45	2D	055
SO	14	0E	016	.	46	2E	056
SI	15	0F	017	/	47	2F	057
DLE	16	10	020	0	48	30	060
DC1	17	11	021	1	49	31	061
DC2	18	12	022	2	50	32	062
DC3	19	13	023	3	51	33	063
DC4	20	14	024	4	52	34	064
NAK	21	15	025	5	53	35	065
SYN	22	16	026	6	54	36	066
ETB	23	17	027	7	55	37	067
CAN	24	18	030	8	56	38	070
EM	25	19	031	9	57	39	071
SUB	26	1A	032	:	58	3A	072
ESC	27	1B	033	;	59	3B	073
FS	28	1C	034	<	60	3C	074
GS	29	1D	035	=	61	3D	075
RS	30	1E	036	>	62	3E	076
US	31	1F	037	?	63	3F	077

ASCII TABLE

ASCII	DECIMAL	HEX	OCTAL	ASCII	DECIMAL	HEX	OCTAL
@	64	40	100	`	96	60	140
A	65	41	101	a	97	61	141
B	66	42	102	b	98	62	142
C	67	43	103	c	99	63	143
D	68	44	104	d	100	64	144
E	69	45	105	e	101	65	145
F	70	46	106	f	102	66	146
G	71	47	107	g	103	67	147
H	72	48	110	h	104	68	150
I	73	49	111	i	105	69	151
J	74	4A	112	j	106	6A	152
K	75	4B	113	k	107	6B	153
L	76	4C	114	l	108	6C	154
M	77	4D	115	m	109	6D	155
N	78	4E	116	n	110	6E	156
O	79	4F	117	o	111	6F	157
P	80	50	120	p	112	70	160
Q	81	51	121	q	113	71	161
R	82	52	122	r	114	72	162
S	83	53	123	s	115	73	163
T	84	54	124	t	116	74	164
U	85	55	125	u	117	75	165
V	86	56	126	v	118	76	166
W	87	57	127	w	119	77	167
X	88	58	130	x	120	78	170
Y	89	59	131	y	121	79	171
Z	90	5A	132	z	122	7A	172
[91	5B	133	{	123	7B	173
\	92	5C	134		124	7C	174
]	93	5D	135	}	125	7D	175
^	94	5E	135	~	126	7E	176
_	95	5F	137	DEL	127	7F	177

ASCII TABLE

DECIMAL	HEX	OCTAL	DECIMAL	HEX	OCTAL
128	80	200	160	A0	240
129	81	201	161	A1	241
130	82	202	162	A2	242
131	83	203	163	A3	243
132	84	204	164	A4	244
133	85	205	165	A5	245
134	86	206	166	A6	246
135	87	207	167	A7	247
136	88	210	168	A8	250
137	89	211	169	A9	251
138	8A	212	170	AA	252
139	8B	213	171	AB	253
140	8C	214	172	AC	254
141	8D	215	173	AD	255
142	8E	216	174	AE	256
143	8F	217	175	AF	257
144	90	220	176	B0	260
145	91	221	177	B1	261
146	92	222	178	B2	262
147	93	223	179	B3	263
148	94	224	180	B4	264
149	95	225	181	B5	265
150	96	226	182	B6	266
151	97	227	183	B7	267
152	98	230	184	B8	270
153	99	231	185	B9	271
154	9A	232	186	BA	272
155	9B	233	187	BB	273
156	9C	234	188	BC	274
157	9D	235	189	BD	275
158	9E	236	190	BE	276
159	9F	237	191	BF	277

ASCII TABLE

DECIMAL	HEX	OCTAL	DECIMAL	HEX	OCTAL
192	C0	300	224	E0	340
193	C1	301	225	E1	341
194	C2	302	226	E2	342
195	C3	303	227	E3	343
196	C4	304	228	E4	344
197	C5	305	229	E5	345
198	C6	306	230	E6	346
199	C7	307	231	E7	347
200	C8	310	232	E8	350
201	C9	311	233	E9	351
202	CA	312	234	EA	352
203	CB	313	235	EB	353
204	CC	314	236	EC	354
205	CD	315	237	ED	355
206	CE	316	238	EE	356
207	CF	317	239	EF	357
208	D0	320	240	F0	360
209	D1	321	241	F1	361
210	D2	322	242	F2	362
211	D3	323	243	F3	363
212	D4	324	244	F4	364
213	D5	325	245	F5	365
214	D6	326	246	F6	366
215	D7	327	247	F7	367
216	D8	330	248	F8	370
217	D9	331	249	F9	371
218	DA	332	250	FA	372
219	DB	333	251	FB	373
220	DC	334	252	FC	374
221	DD	335	253	FD	375
222	DE	336	254	FE	376
223	DF	337	255	FF	377

OPCODE CHART

L O W	High Nybble							
	8	9	A	B	C	D	E	F
0	LET	FN	CLS	STEP		(<=
1	FOR	DEF		TO			^	<>
2	PRINT	!		THEN		ATN	*	
3	NEXT	ON		TAB		FILESIZE	+	
4	IF	OUT		ELSE	SQRT	FILEPTR		<
5	READ	FILL		CHR\$		ADDR	-	=
6	INPUT	EXIT	DOSCMD	ASC	INT	INSTAT		>
7	DATA	OPEN	APPEND	VAL		OUTSTAT	/	NOT
8	GOTO	CLOSE		STR\$		FREE		
9	GOSUB	WRITE		NOENDMARK		INP	NOTE2	
A	RETURN	NOTE1		INCHAR\$	SGN	EXAM		
B	DIM	CHAIN		FILE	SIN	ABS		
C	STOP	LINE			LEN	COS	AND	
D	END	DESTROY			CALL	LOG	OR	
E	RESTORE	CREATE			RND	EXP		
F	REM	ERRSET				TYP	>=	

NOTE1--9A indicates the next two bytes are a binary line number.

NOTE2--E9 is used internally as negate

DOSCMD available in MicroDoZ version only.

INSTAT and OUTSTAT available in MicroDoZ and CP/M versions only.

ERRATA 1
CP/M baZic CAUTION

To be compatible with non-CP/M versions of baZic and with North Star BASIC, the "DESTROY" statement destroys all files of the specified name on the specified disk.

For example:

If you have CP/M file on Drive B:

TEST.COM
TEST.SRC
TEST.LST
TEST.003

DESTROY "TEST,2" will wipe out all of these files.

REMEMBER: Your file names must be unique when using baZic.

ERRATA 2

The following topics are not described properly in the manual:

1. 3.4.4 CLOSE a channel

The manual states that the channel will be closed on a STOP or a Control C. In fact the channel is not closed, but the buffer is flushed to the disk to preserve any data you may have stored in the buffer.

2. 4.1.4 LOGarithmic value

The last example has a typographical error. The example should read:

C=LOG(23.14069) (C=3.1415925)

3. 9.2 XREF

The Printer option under XREF prints only to the right Horizon serial port. No provision has been made to print to the parallel port or any other port.

In some early printings of the manual, Listing 1 shows the variable "O" (the letter O) as the number "0" (zero). This listing should be changed on lines 30, 70-110, and 130-150 to reflect the variable "O."

4. Common MEMSETs

The manual does not list the correct MEMSET locations for the MicroDoZ and CP/M versions of baZic. The common MEMSET locations are:

Memory Size	Address in Decimal	
	MicroDoZ	CP/M
48K	43519	42245
56K	51711	50437
60K	56319	54533

5. Auto byte for CP/M baZic

The Auto byte for CP/M baZic is 271 decimal. This byte should be FILLED with a 0 to activate the auto start feature. The general procedure is to SCRatch any previous program, LOAD or enter your turnkey program, FILL 271,0 to activate the turnkey mode, and BYE to CP/M to save baZic as a file.

ERRATA 3
NOTES ON CP/M BAZIC 03/03

Version 03/03 of CP/M baZic has now been released. This version fixes several bugs or problems that were found in 02/02. In addition, a new feature has been added.

CP/M baZic now has a new function which is called CPMFN (CP/M Function). This function has been added to allow the baZic programmer to call any of the CP/M system function calls. The command syntax is:

T=CPMFN(<C ARGUMENT> [,<DE ARGUMENT>])

The function is called by passing the function number as the first argument to the function call. The function numbers are listed in the back of most CP/M documentation manuals. The value passed as the first argument will be "inserted" in the C register when the system call is made.

Many system calls require a second argument which is passed to the DE register pair. This argument is optional to the CPMFN, but if the system call requires a second argument, you must include this value in the CPMFN call.

When the function returns, the variable set equal to the function call (T in the example) will be equal to the value of the HL register pair.

The CREATE statement now creates a dummy file the size of the size variable, if it is specified. This means that the space specified in the CREATE statement is reserved on the disk when the CREATE statement is executed.

baZic now updates the file directory when a program is exited. Previously, under some conditions, the file directory was not updated, leading to errors in the files.

baZic programs can now be CLOSED without previously OPENING the file.

The implementation of the cursor positioning and clear screen statements have been changed at the beginning of baZic to allow more CRTs to be configured. Two modes are now supported for setting the cursor addressing and clear screen sequence. At the beginning of the cursor addressing and clear screen sequence is a byte which controls the mode. If the byte contains a 0FFH, baZic will assume a machine language routine is in place starting at the next location after the mode byte and will jump to this location and execute the code located there.

If the mode byte is any other value, baZic will assume that the byte represents the number of codes to output and will output that number of bytes, starting with the first byte following the mode byte. The string of codes in this mode must end with a "\$" code (36 Decimal or 24 Hex). The SOROC option of the CRT program uses this mode while all other terminals in the CRT program use the first mode.

The HAZELTINE option uses the first mode described but is slightly different in that the HAZELTINE requires the cursor addressing codes be issued column,row whereas most CRTs require a row,column sequence. This sequence is changed for the HAZELTINE option by reversing the bytes at location 0131H and 0138H. The MOV E,L instruction at 0131H is changed to a MOV E,H and the MOV E,H at 0138H is changed to a MOV 3,L. This reverses the order the cursor addressing codes are output.

The CRT program must be run on a "virgin" copy of baZic as delivered on the master diskette. You cannot configure a copy of baZic for the SOROC and then run CRT again to configure the same copy for a Z19. You must use a copy from the master diskette to configure for each different CRT you might have. See the enclosed source listing for more information.

PARTIAL SOURCE LISTING FOR CP/M BAZIC 03/03

```

0100 AF          ENTRY1 XRA A          Clears program
0101 18 02      JR RENTR1
0103 00          MEMFLG DB 0          Auto MEMSET if 1
0104 37          ENTRY2 STC          Saves program
0105 21 3560    RENTR1 LXI H,ENDBAS
0108 11 8000    LXI D,08000H          End memory
010B C3 0169    JMP INTL1
                ***
010E 50          LINECT DB 50H        Initial line length
010F 01          AUTOS DB 1          Auto-start flag
0110 E8          PROM DB 0E8H        Disk controller origin
0111 29DE       LINETB DW PHPOS       Print head table
0113 18          PAGES DB 18H        CRT Lines per page
                *
0114 C3 0264    ENTRY3 JMP READY1     Saves data
                *
0117 08          BSPC DB 8H          Character delete
0118 00          CONC DB 0          Control C inhibit flag
    
```

Cursor addressing routine. The example below is for the Zenith Z19 CRT. On entry register D is the column and register E the line.

0119 FF	GOTOYX DB OFFH	Indicates assembly language routine follows.
011A D5	PUSH D	
011B 0E 06	MVI C,6	
011D 1E 1B	MVI E,27	
011F CD 2935	CALL BDOSC	Use this address for BDOS calls to save the IX & IY if you have a CP/M look-a-like that uses the IX & IY registers.
0122 0E 06	MVI C,6	
0124 1E 59	MVI E, "Y"	
0126 CD 2935	CALL BDOSC	
0129 2A 0144	LHLD OFFSET	
012C D1	POP D	
012D 19	DAD D	
012E E5	PUSH H	
012F 0E 06	MVI C,6	
0131 5D	MOV E,L	
0132 CD 2935	CALL BDOSC	
0135 E1	POP H	
0136 0E 06	MVI C,6	
0138 5C	MOV E,H	
0139 CD 2935	CALL BDOSC	
013C C9	RET	
013D 0007	DS 7	
0144 1F1F	OFFSET DW 1F1FH	Zenith Z19 offset

Clear screen routine. The example below is for the Zenith Z19 CRT.

0146 FF	CLS1 DB 0FFH	Direct console output Z19 clear screen sequence is ESC "E"
0147 0E 06	MVI C,6	
0149 1E 1B	MVI E,27	
014B CD 2935	CALL BDOSC	
014E 0E 06	MVI C,6	
0150 1E 45	MVI E, "E"	
0152 CD 2935	CALL BDOSC	
0155 C9	RET	
0156 0003	DS 3	
0159 34AF	SBDATA DW ATN	Addresses used by SHORTB program.
015B 33B0	DW COS	
015D 32A0	DW LOG	
015F 3169	DW EXP	
0161 3125	DW TOPWR	
0163 308E	DW SQRT	
0165 3042	DW RND1	
0167 3167	DW TOPWR4+1	

*

```

*
INITIALIZATION
0169 3E 01 INTL1 MVI A,1
016B 32 2D2B STA CHAINFG
016E 3E FF MVI A,0FFH
0170 32 2B31 STA CRLFLG
0173 23 INX H
0174 22 2D3C SHLD BEGPRG
0177 ED 53 2D3E SDED ENDMEM
017B E3 00 MVI A,0
017D 32 2B36 STA APNDFLG
0180 31 29D5 INTL2 LXI SP,PRPORT Re-entry from LOAD
0183 ED 73 2B38 SSPD STKSET
0187 3E 01 MVI A,1
0189 32 2B33 STA RUNFLG
018C 3E 00 MVI A,0
018E 32 29D5 STA PRPORT
0191 32 2B2E STA AUTON
      0000 IF HARDFP
      CALL WAKEUP
      FFFF ELSE
0194 0003 DS 3
      ENDIF
0197 30 4C JRNC INTL8
*

```