baZic Operator's Manual

Release II

developed for use with the

CP/M,$^{(R)}$

NorthStar^{TM} DOS

and

MicroDoZ$^{(R)}$

Operating Systems

Revisions 05/04
January 5. 1983

developed by

Micro Mike's, Inc.
814 S Lamar
Amarillo, TX 79106 USA

telephone: 806-372-3633

making technology uncomplicated ... for People^{SM}

Copyright 1980, 1981, 1982 by Micro Mike's, Inc.

Unauthorized duplication and/or transfer of this software or manual is illegal.

COPYRIGHT NOTICE

Copyright (c) 1982 by Micro Mike's, Inc. All rights reserved. No part of this publication or associated programs may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Micro Mike's, Inc., 3Ø15 Plains Blvd., Amarillo, Texas 791Ø2.

This program package is licensed for use on one (1) CPU.

DISCLAIMER

Micro Mike's, Inc. makes no representation or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Micro Mike's, Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Micro Mike's, Inc. to notify any persons of such revision or changes.

TRADEMARK NOTICES

This document will mention several names and products which have been granted trademarks. It is the intent of Micro Mike's, Inc. to acknowledge and respect the trademarks of these companies so that all the rights and privileges of these companies are preserved.

baZic(R) and **MicroDoZ**(R) are registered trademarks of Micro Mike's, Inc.

CP/M(R) and MP/M(R) are registered trademarks of Digital Research Corp.

Z8Ø(R) is a registered trademark of Zilog, Incorporated.

CLS and !@

For CLS and !@ to function properly, an ASCII file called CONFIG.SYS must reside in the root directory on the boot disk. This file should contain the following:

```
DEVICE = ANSI.SYS
```

The file ANSI.SYS must also reside in the root directory of the boot disk. The ANSI.SYS file is found normally on the MS-DOS (or PC-DOS) boot disk or root directory of the hard disk.

For RTNX multi-user systems, an ASCII file called CONFIG.SLV must reside in the root directory of the boot disk. It should contain the following:

```
DEVICE = ANSI.SYS
```

!CHR$(27)+"[2J"
will also clear the screen.

BYE
BYE is now both a statement and command.

BREAK <numeric expression>
If the value of the expression is 0 Control C is disabled; otherwise Control C is enabled. BREAK is both a statement and a command. This release of baZic86 runs programs considerably faster with Control C disabled.

DESTROY <file name> [,<file type>]
If file type is not specified an extension of .003 is assummed.

CHDIR <string expression>
Changes the DOS current directory of the specified or default drive.

X = CALL (<memory address> [,<argument>])
Memory address must be a numeric expression with a decimal value in the range 0 to 1,048,575. The optional argument must be a numeric expression with a value in the range 0 to 4,294,836,225. It is passed as a 32-bit binary number with the high word in dx and the low word in ax. The return value is assumed to be a 32-bit binary number with the high word in dx and the low word in ax. The ds, es ss, and sp must be restored to the entry condition before returning. MUST USE A FAR RETURN.

ERRSET

New error codes have been added to give more information on critical errors. For disk errors 20 is added to the error code returned by DOS. For character device errors 40 is added to the error code returned by DOS.

CPMFN

The CPMFN statement/command is not functional under baZic86.

DOSCMD

The DOSCMD statement/command of the MicroDoZ version of baZic is not functional under baZic86.

DOS critical error codes (INT 24)

0	Attempt to write on write-protected diskette
1	Unknown unit
2	Drive not ready
3	Unknown command
4	Data error (CRC)
5	Bad request structure length
6	Seek error
7	Unknown media type
8	Sector not found
9	Printer out of paper
10	Write fault
11	Read fault
12	General failure

For example, an error code of 22 indicates drive not ready. An error code of 49 indicates printer out of paper.

TABLE OF CONTENTS

INTRODUCTION

baZic, one of the fastest BASIC interpreter languages available for the 8-bit microprocessor, was written to utilize the full Z80 instruction set.

Originally developed to provide a NorthStar-compatible BASIC interpreter which allows execution of NorthStar BASIC programs with little or no modifications, versions of Release II of baZic will operate under NorthStar DOS, CP/M(R) and Micro Mike's, Inc. Disk Operating System (MicroDoZ.) All three versions are considered separate products. With the addition of the MicroDoZ and CP/M versions, **baZic** becomes highly transportable for software developers.

Some of the advances of **baZic** over NorthStar BASIC include much faster execution of BASIC programs and increased power by the addition of new statements and functions.

All NorthStar features are supported through Release 5.2 of NorthStar BASIC. However, **baZic** will not run programs written with BASIC 5.2 which use the FILESIZE or FILEPTR functions because 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.

In the MicroDoZ version, all MicroDoZ commands can be executed from **baZic** as a statement or command.

References to NorthStar BASIC in this manual will be made as 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, except in those few cases where programs use FILLs or CALLs to BASIC locations which will be different or unnecessary under **baZic**.

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, 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 Z8Ø 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 NorthStar 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. (See the Beginner's Guide to **baZic**, Book I by Micro Mike's, Inc.)

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. In most cases, examples will follow the description.

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
TYPEXPR	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. Some 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 NorthStar 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, function or line number.) 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. Line numbers are converted to a three-byte code. The first byte signifies that this is a line number. The second byte is the least significant byte of the line number and the third is the most significant 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.

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** (Ø-7) except the CP/M version which supports three devices (Ø-2). A number sign (#) always precedes a device number in a **baZic** program. The device numbers are defined in the appropriate disk operating system (MicroDoZ, CP/M or NorthStar DOS.)

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. MDZ/OS or JOESHARE II, 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. Drives 1-7 correspond to CP/M Drives A-G.

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 (Ø-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 (".ØØ3", ".ØØ2", etc.).

File Type numbers are used to distinguish between different kinds of files. File types may range from Ø to 63 under MicroDoZ. Under CP/M the range is Ø 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 closely as possible by defining the type extension such that: .ØØ1 is for assembly language, .ØØ2 is for a **baZic** program and .ØØ3 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 Ø to 65535, etc.), filenames (must have eight letters or fewer,) drive number specifications (Drives 1-7), etc.

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 Ø 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, enclosed 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 value when evaluated.

Sectors can be any value and are hardware dependent. Double density and quad capacity NorthStar disk drives have 512 bytes per sector while single density NorthStar 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 Ø to 9. String variable names are similar except a dollar sign ($) is placed after the letter and number to show the variable is a string variable. Arrays have an expression enclosed in parentheses () immediately following the letter or number. Variables can and do change their values many times during the execution of a program. A numeric variables, a string, an array, a user-defined function can have the same name and not be related (e.g. A9, A9$, A9(1) and FNA9(1).)

This page left blank intentionally.

CONFIGURING baZic

Before configuring baZic for your system, the registration card should be filled out and returned immediately to Micro Mike's, Inc. A copy should be made of the master, using a gross copy program supplied with the operating system. See your operating system manual for instructions for copying diskettes.

The master diskette should be stored in a safe place. All updates will be supplied only with the return of the master diskette to Micro Mike's, Inc., and only if the registration card has been returned and that serial number is on file.

2.1 Configuring baZic for MicroDoZ

A configure program is delivered with MicroDoZbaZic which should be run before attempting to use MicroDoZbaZic version. Consult the MicroDoZ manual for instructions. As delivered, MicroDoz boots first to a copy program and then to a configure program.

2.2 Configuring baZic for CP/M

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 a 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.Ø or a more recent version. The CP/M operating system prompt will be displayed as follows:

```
A>
```

2.2.1 Set Upper Memory Limit, Turnkey Command

baZic for CP/M, Release Ø5/Ø4 and later, finds its own MEMSET level at bootup time. Releases Ø3/Ø3 and earlier require the following MEMSET procedures:

In the following example, we will turnkey and set upper memory limits for 1Ø-digit software floating point baZic. We will load 1Ø-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 "BAZIC1Ø.COM," 54 blocks long, on the disk to preserve these changes.

Common MEMSET values for baZic for CP/M:

48K -- 42245	6ØK -- 54533
56K -- 5Ø437	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 5Ø437 <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.) Under some floppy disk systems, the disk in Logged Drive A must be sysgened. The copy of the baZic distribution disk must be in Drive A to continue with the configuration.

Following is an example configuration of 1Ø-digit software floating point baZic, 56K CP/M (your responses are underlined):

```
A>^C
A>BAZIC1Ø <CR>
READY
FILL 271,Ø <CR>
READY
1Ø CHAIN "MENU" <CR>
MEMSET 5Ø437 <CR>
READY
BYE <CR>
A>SAVE 54 BAZIC1Ø.COM <CR>
A>
```

2.2.2 CRT Configuration

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.ØØ2." In the following example we will configure 1Ø-digit software floating point baZic for a SOROC IQ 12Ø terminal (your responses are underlined):

```
A>BAZIC10 <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=DIGITAL VT-100
8=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 (those versions of baZic with an "F" in the file name, e.g. BAZIC08F.COM) 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 loops (prolate cycloid or loop-de-loops) will appear on the screen. Below the 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
```

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.

2.2.3 Adapting baZic for MP/M I

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
```

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

```
READY
FILL 28Ø,1 <CR>
READY
```

2.2.5 Adapting baZic for MP/M II

Control C must be disabled (FILL 28Ø,1, as in the MP/M I example above).

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

```
Ø319 Hex
1CØØ Hex
1F24 Hex
```

2.2.6 baZic File Name Extensions

Under CP/M and MP/M, baZic programs will have a ".ØØ2" extension, e.g., "PROGRAM.ØØ2." baZic data files generally will have a ".ØØ3" extension, unless specified otherwise. Type 1 machine language programs will have a ".ØØ1" extension.

2.3 Configuring baZic for NorthStar DOS

CRT is the program used to configure the 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.

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 fewer, 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= 1Ø 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= 1Ø 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 for additional information.

Selecting the "OTHER" option will result in the program asking 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 "Ø" 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 "Ø" 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.

2.4 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 MDZ/OS PROGRAMMER manual for more information on configuring a CRT for MicroDoZ.

This page left blank intentionally.

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 4 (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)
```

3.1 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.

3.1.1 List a program (LIST)

```
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 (Ø-7) preceded by the number sign (#). If the DEVICE# is not used, the default device is printed to (generally device Ø.)

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)
```

3.1.2 Delete Line Numbers (DEL)

```
     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 and existing 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 either of the specified line numbers does 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)
     10<CR)                 (DELete Line 10)
```

3.1.3 Scratch a Program (SCR)

```
     SCR
```

The SCRatch command is issued to cause the current program and all its associated variables to be SCRatched from internal memory. After a program is SCRatched, it cannot be recovered from RAM. If you want to use the program 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.

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 1ØØ               (LIST line number 1ØØ only)
     LIST 1ØØ,              (LIST line number 1ØØ to end)
     LIST #2                (LIST all line numbers to Device #2)
     LIST #2,1ØØ,5ØØ        (LIST lines 1ØØ to 5ØØ on Device #2)
```

3.1.2 Delete Line Numbers (DEL)

```
     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 and existing 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 either of the specified line numbers does not exist, a LINE NUMBER ERROR will be generated. The value of line numbers must always be in the range of Ø 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 1Ø,1ØØ             (DELete Lines 1Ø through 1ØØ)
     DEL 1,2                (DELete Lines 1 and 2)
     1Ø<CR)                 (DELete Line 1Ø)
```

3.1.3 Scratch a Program (SCR)

```
     SCR
```

The SCRatch command is issued to cause the current program and all its associated variables to be SCRatched from internal memory. After a program is SCRatched, it cannot be recovered from RAM. If you want to use the program 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.

3.1.4 Renumber a Program (REN)

```
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 now present within the current program, these references will not be changed. 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.

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 1Ø and automatically incrementing by 1Ø. The LINE# argument must be a positive integer in the range of Ø 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 1Ø, increment by 1Ø)
REN 1ØØ,2Ø             (Start with 1ØØ, increment by 2Ø)
REN 15                 (Assumes INCREMENTAL VALUE of 1Ø)
```

3.1.5 Automatic Line Numbering (AUTO)

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

This command causes baZic to generate line numbers AUTOmatically as the programmer is entering 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 1Ø and increment by 1Ø. If no incremental value is specified, baZic will start at the number specified and increment by 1Ø.

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 Ø 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)
```

3.1.6 Program Size (PSIZE)

```
     PSIZE
```

This command causes baZic to return the size in blocks (256-byte blocks) 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.

3.1.7 Set Memory Upper Limits (MEMSET)

```
     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.

Various MEMSET values include:

	NorthStar DOS	MicroDoz	CP/M
32K	32767		
48K	49151	43519	42245
56K	57343	51711	50437
60K		56319	54533
64K			58373

Under CP/M, the MEMSET level may vary according to a computers particular implementation of CP/M.

To find a computer's MEMSET level, enter one of the following formulas:

```
     for MicroDoZ -- !EXAM (1) + 256 * EXAM (2)-1
     for CP/M     -- !EXAM (7) * 256 + EXAM (6)-1
```

Release Ø5/Ø4 of baZic finds its own MEMSET level when loading into memory.

3.2 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.

3.2.1 Catalog a Disk (CAT)

```
     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 (Ø-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 #Ø). 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 under which **baZic** can run. 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 (Ø to 127 Decimal)
GO Address (If Type 1 in Hex)
R/O if Read Only File
SYSTEM if System File
Attribute Field (Ø to 63 Decimal)

A sample CATalog under MicroDoZ would appear as follows:

```
READY
CAT

MICRODOZ     4    2Ø D   Ø                      AF  Ø
M2DØØM      14    1Ø D   1 2DØØ                 AF  Ø
TEST        19     2 D   2       R/O SYSTEM AF  Ø
BAZIC       2Ø    54 D   1 Ø1ØØ                 AF  Ø
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:

```
1Ø 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

: BAZICØ8F COM : BAZIC1ØF COM : BAZIC12F COM : BAZIC14F COM
: CRT      ØØ3
```

3.2.2 Save a Program (SAVE)

```
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
```

3.2.3 Save a New Program (NSAVE)

```
    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 supports dynamic file alloction 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 a 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)
```

3.2.4 Load a Program (LOAD)

```
    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         (Load "CONTROL" from default drive)
```

3.2.5 Bye to Disk Operating System (BYE)

```
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 Ø1ØØ Hex.

The following table describes the three entry points for **baZic**.

Hex Address	Entry characteristics
ORG	Clears program and all variables
ORG + Ø4 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.

3.3 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.

3.3.1 Run a Program (RUN)

```
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 LINE# must be a legal and existing line. If the LINE# does not exist, a LINE NUMBER ERROR will be displayed.

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                 (Program starts execution at first line)
RUN 5ØØ             (Program starts at LINE 5ØØ)
```

3.3.2 Continue a Program (CONT)

```
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. A CONT ERROR message will be displayed if a Control C is issued during an INCHAR$(Ø) statement in a program.

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

3.3.3 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 4.4.9) and APPEND (Section 4.4.8) for more information.

This page left blank intentionally.

STATEMENTS

Statements are the "stuff" programs are made of. To define a program to do any useful task, the task must be divided into 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 control the way the statement is executed.

4.1 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.

4.1.1 Data Constants (DATA)

```
     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 slightly. 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"
```

4.1.2 Read Data Constants (READ)

```
     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
```

4.1.3 Restore Data Pointer (RESTORE)

```
     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 4.1.2 (READ data constants) shows the use of the RESTORE statement.

4.2 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.

4.2.1 Print a Variable (PRINT)

```
     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. Under baZic for CP/M, PRINT #Ø is the console direct device, PRINT #1 is the list device and PRINT #2 is the punch device.

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 and line feed 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. See Section 5.5.4.

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
```

4.2.1.1 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. This will occur only in the default format. In other formats, a FORMAT ERROR will occur.

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. If the number is too large, a FORMAT ERROR will occur.

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

4.2.1.2 Print at (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 IOEDIT 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.

An example of this print function is illustrated with a "CRT" that has 10 lines and 20 spaces per line. If the following program were RUN on this CRT, the output would appear as follows:

```
10 PRINT@(2,15),"*"

--------------------
|                  |
|              *   |
|READY             |
|                  |
|                  |
--------------------
```

4.2.2 Input a Variable (INPUT)

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

The INPUT statement is used to input a value from the user of the program and assign this value to a variable. If no DEVICE# is specified, the input is taken from the default device. If the DEVICE# is specified it must be preceded by the number sign (#) and must be a legal device number (0 to 7). Under baZic for CP/M, INPUT#0 is the console direct device, INPUT#1 is the reader device.

The STRING PROMPT argument is optional but can be used to specify a prompt string to the user. This string constant (delimited by quotation marks) will be printed before taking the input and will suppress the normal question mark (?) that is printed as the prompt when the STRING PROMPT argument is not passed to the INPUT statement.

The VARIABLE LIST can be one or more variables to be "filled" by the user's response. If only one numeric or string variable is to be input, the user enters the value and presses RETURN to signify the value has been entered.

If more than one numeric variable is specified, the user must separate each value entered by a comma. If the user fails to enter all variables asked for, baZic will prompt the user with two question marks (??). Another RETURN terminates the input. More than one string variable cannot be entered by this method.

If a number and a string are to be input from the same INPUT statement, the numeric variable should be input first. Otherwise the comma and the number will be input as part of the string since baZic has no way of knowing the end of the string input.

If the input is to be numeric and a string is entered, baZic will respond with an INPUT ERROR -- PLEASE RETYPE. If the input is to be a string input, a carriage return only is acceptable and results in a null string for the specified variable.

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$
```

4.2.3 Input a Variable (INPUT1)

```
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.

4.2.4 Output a Byte (OUT)

```
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.

4.3 Branching Statements

In baZic, the "normal" flow of processing is from the smallest line number to the largest line number. This situation is rarely efficient 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.

4.3.1 Go to a Line Number (GOTO)

```
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.

Because 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
```

4.3.2 Go Subroutine (GOSUB)

```
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
```

4.3.3 Return from Subroutine (RETURN)

```
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 7.2 for more details. A RETURN also "EXITs" all FOR NEXT loops within the subroutine. See Section 4.3.7 for more information.

4.3.4 On Value Go to Line Number (ON GOTO)

```
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 based on the results of an expression. The numeric expression (#EXPR) is evaluated to determine the line number in the list of line numbers to which 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)
```

4.3.5 On Value Go Subroutine (ON GOSUB)

```
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 statement 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
READY
RUN

THIS IS SUBROUTINE ONE
THIS IS SUBROUTINE TWO
THIS IS SUBROUTINE THREE
READY
```

4.3.6 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 also can 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 9.3.2 (IF THEN Evaluation) for more information.

IF THEN ELSE statements can be nested, but care should be exercised when doing so.

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." (See Section 6.2) 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"
```

4.3.7 FOR NEXT STEP EXIT

```
FOR <CONTROL>=<INITIAL> TO <LIMIT> [STEP <VALUE>]
NEXT [<CONTROL>]
EXIT <LINE#>
```

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 INITIAL VALUE of 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 Ø 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 Ø 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
```

4.4 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 NorthStar convention. All baZic program files must have a ".ØØ2" extension and all data files must have a ".ØØ3" extension (unless the program specifies another numeric type).

The NorthStar 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.

4.4.1 Create a File (CREATE)

```
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 is in 256-byte blocks and should be an even number. It must be within the range of available space on your disk drives. This value can be as much as 16 megabytes (16,776,96Ø bytes) on a hard disk system but is hardware dependent.

The FILETYPE argument can be any positive integer number in the range of Ø 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 and a dummy file the size of the variable, if it is specified, is created. This means that the space specified in the CREATE statement is reserved on the disk when the create statement is executed.

Examples of the CREATE statement are:

```
CREATE "DATAFILE,2",1ØØ
1Ø CREATE "MENU1",2Ø,2       (Create file for baZic program)
2Ø CREATE N$+D$,N1,N2
```

4.4.2 Destroy a File (DESTROY)

```
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 or the ERA command in CP/M. The only action taken is the removal of the specified file name from the directory of its disk. The actual file is not changed.

If the specified file does not exist, an ARGument error is returned.

Examples of the DESTROY statement are:

```
     DESTROY "DATAFILE,2"
     1Ø DESTROY "MENU1"
     2Ø DESTROY N$+D$
```

Caution: To be compatible with other versions of **baZic** and with BASIC, the DESTROY statement in the CP/M version 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.SRM
     TEST.LST
     TEST.ØØ3
```

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

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

4.4.3 Open a Channel (OPEN)

```
     OPEN #<CHANNEL#>[%<TYPEXPR>],<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 TYPEXPR 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 TYPEXPR allows a **baZic** program to open any file, including **baZic** programs and machine language programs. The OPEN will be successful only if the TYPEXPR 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 at a time may be 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$+D$
```

4.4.4 Close a Channel (CLOSE)

```
     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.

A program can CLOSE a channel without having previously OPENed the channel.

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 statement.

The program terminates execution because of an error.

If a program encounters a STOP statement or a control C is entered by the user, the buffer is flushed but the channel still remains "OPEN."

Examples of the CLOSE statement are as follows:

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

4.4.5 Read a File Variable (READ)

```
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 Ø 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 Ø 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 Ø to 255, the range of a byte value.

Examples of the use of the READ statement follow:

```
READ #1,A,B,C
1Ø READ #1%512*N,A,B,B$
2Ø READ #1,&A,&B,&C
```

4.4.6 Write a File Variable (Write)

```
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 Ø 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 value must be a positive integer number and must be smaller than the number of bytes in the file.

The file pointer can be set to the beginning of the file by issuing a random write to position Ø followed by a NOENDMARK. The fourth example demonstrates this feature.

WRITE(ing) 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 bytewise. These variables must have a byte value in the range of Ø to 255.

Examples of the use of the WRITE statement are:

```
     WRITE #1,A,B,C
     1Ø WRITE #D%512*N,A,B,C,B$
     2Ø WRITE #4%256,&A,&B,&C,NOENDMARK
     3Ø WRITE #1%Ø,NOENDMARK        (Position pointer to beginning
                                            of file)
```

4.4.7 No End Mark (NOENDMARK)

```
     NOENDMARK
```

When baZic WRITEs to a file, it places an ENDMARK after the data written. The NOENDMARK command generally is used in random writes to avoid the loss of the following record.

Examples of the NOENDMARK can be found in Section 4.4.6.

4.4.8 Append a Program (APPEND)

```
     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 deleted and the lines in the specified program (FILENAME) will be tacked on to the loaded program.

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<>Ø), the GOTO will be executed.

```
3 APPEND 1ØØØ,A$
3 APPEND "CSUB"
1Ø IF A=Ø THEN APPEND A$ ELSE GOTO 5ØØ
```

4.4.9 Chain to a Program (CHAIN)

```
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 CLOSEd automatically.

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"
1Ø CHAIN A$+B$
```

4.4.1Ø DOS Command (DOSCMD)

```
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.

4.5 Miscellaneous Statements

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

4.5.1 Dimension a Variable (DIM)

```
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 1Ø bytes. Multiple DIMensions can be performed on the same program line by separating each variable by a comma.

A numeric array variable is DIMensioned automatically to 11 (elements Ø to 1Ø) 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 be simulated easily under baZic. Strings can be any length, limited only by the amount of internal memory available. A string is DIMensioned automatically to 10 bytes (or characters) simply by 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. All of the elements within a numeric array are set to 0.

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)
```

4.5.2 Remark a Comment (REM)

```
     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 (SET TO 20) IS THE NUMBER OF RECORDS TO PRINT PER PAGE

     30 REM A SHORT REMARK\X=20\REM X WILL NOT BE SET TO 20
```

4.5.3 Assign a Variable (LET)

```
       [ 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 destination 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)
```

4.5.4 CLear the Screen (CLS)

```
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 IOEDIT 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
```

4.5.5 Fill a Memory Location (FILL)

```
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 the operating system 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
     1Ø FILL 124Ø5,2Ø1
```

4.5.6 Set Error Trapping (ERRSET)

```
     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 8.1 (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:

```
     1Ø ERRSET 2ØØ,E1,E2
     2Ø ERRSET
```

4.5.7 Line Length (LINE)

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

The LINE statement is used to set the line length of any device. The default line length is 8Ø characters. The line length can be set for any value from 1Ø to 165 characters. The DEVICE# must be a legal device in your system and must evaluate to a positive integer from Ø to 7.

Normally, **baZic** echos a carriage return and line feed automatically when the line length is reached while printing to a device number. If this is not desirable, the automatic carriage return and line feed can be suppressed by passing an additional optional numeric expression which evaluates to zero (Ø). 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
20 LINE #1,132,0
```

4.5.8 Stop a Program (STOP)

```
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.

The STOP command is a powerful debugging tool because it halts program execution at a specific place and allows variables to be examined and modified.

Examples of the use of the STOP statement are:

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

4.5.9 End a Program (END)

```
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.

5.1 Math Functions

All of the following functions are related to mathematical calculations.

5.1.1 Absolute Value (ABS)

```
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)
```

5.1.2 Sign of a Number (SGN)

```
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 Ø if the value of the expression is zero (Ø).

Examples of the SGN function follow:

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

5.1.3 Integer Value (INT)

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

5.1.4 Logarithmic Value (LOG)

```
LOG(<NUMERIC EXPRESSION>)
```

This function returns an approximation (accuracy is 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 Ø).

Examples of the LOG function are:

```
A=LOG(1)            (A=Ø)
B=LOG(.5)           (B=-.69314717)
C=LOG(23.14Ø69)     (C=3.1415925)
```

5.1.5 Exponential Value (EXP)

```
EXP(<NUMERIC EXPRESSION>)
```

This function returns an approximation (accuracy is 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(Ø)               (A=1)
B=EXP(-.69314717)      (B=.5)
C=EXP(3.1415925)       (C=23.14Ø69)
```

5.1.6 Square Root (SQRT)

```
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 (Ø). 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(Ø)              (A=Ø)
B=SQRT(4)              (B=2)
C=SQRT(3)              (C=1.732Ø5Ø8)
```

5.1.7 Sine (SIN)

```
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:

```
Ø=SIN(Ø)
1=SIN(3.1415926/2)
```

5.1.8 Cosine (COS)

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

5.1.9 Arctanget (ATN)

```
     ATN(<NUMERIC EXPRESSION>)
```

This function returns an approximation of the trigonometric arctangent 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)
```

5.2 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.

5.2.1 Length of a String (LEN)

```
     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 (Ø).

Examples of the LEN function follow:

```
     1Ø DIM A$(2Ø),B$(2Ø),C$(2Ø)
     2Ø A$="MICRO MIKE'S"
     3Ø B$=""
     4Ø PRINT "THE LENGTH OF A$ IS",LEN(A$)
     5Ø PRINT "THE LENGTH OF B$ IS",LEN(B$)
     6Ø PRINT "THE LENGTH OF C$ 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 Ø
     THE LENGTH OF C$ IS 2Ø
     READY
```

5.2.2 Character String (CHR$)

```
     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 Ø to 255. See APPENDIX A 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")
```

5.2.3 ASCii Value (ASC)

```
     ASC(<STRING EXPRESSION>)
```

The ASCii function is the inverse of the CHR$ function. The ASC function returns the ASCII value of 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)

     1Ø A$="ABC"
     2Ø A=ASC(A$(2))      (A=66)
```

5.2.4 Value (VAL)

```
     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("ØØSTRING")    (A=Ø)
     B=VAL("1Ø")          (B=1Ø)
```

5.2.5 String (STR$)

```
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")
```

5.3 Input Functions

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

5.3.1 Input a Character String (INCHAR$)

```
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)
```

5.3.2 Input a Byte (INP)

```
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)
```

5.3.3 Input the Status (INSTAT)

```
     INSTAT(<DEVICE#>)
```

The INSTAT function is available only in the MicroDoZ version 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 (Ø), 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."

5.3.4 Outstatus (OUTSTAT)

```
     OUTSTAT(<DEVICE#>)
```

The OUTSTAT function is available only in the MicroDoZ version 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 (Ø).

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.

5.4 File Functions

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

5.4.1 Type of File Pointer (TYP)

```
     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 functions returns a "Ø," 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 Ø, 1, or 2 by coincidence and not be pointing at an endmark, string, or number.

5.4.2 File Type (FILE)

```
     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 Ø to 127.

Examples of the FILE function follow:

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

5.4.3 File Size (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 256-byte blocks.

An example of the FILESIZE function follows:

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

5.4.4 File Pointer Position (FILEPTR)

```
     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)
```

5.5 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.

5.5.1 Random (RND)

```
RND(<#EXPR>)
```

The RaNDom function returns a psuedo-random value between Ø 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 Z8Ø refresh register for the seed.

If the NUMERIC EXPRESSION evaluates to a zero (Ø), 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 "Ø" argument.

The random seed tends to cycle. Use a -1 at the beginning of the program and zeros thereafter instead of -1's exclusively.

Examples of the RND function follow:

```
A=RND(-1)          (baZic should "pick" the seed)
B=RND(.Ø998)       (seed is .Ø998)
C=RND(Ø)           (use previous number to generate next #)
```

5.5.2 Examine Memory (EXAM)

```
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 Ø and 65535 and must be a Decimal number. The value returned from the function call will be a byte value in the range of Ø to 255.

Examples of the EXAM function follow:

```
1Ø A=EXAM(124Ø5)
2Ø IF EXAM(124Ø5)=2Ø1 THEN D2=124Ø5
3Ø PRINT EXAM(N)
```

5.5.3 Free Memory (FREE)

```
     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(Ø)
```

5.5.4 Tabulate (TAB)

```
     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 #EXPR 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:

```
     1Ø PRINT TAB(2Ø),"THIS IS POSITION 2Ø"
     2Ø IF X<>Y THEN !TAB(1Ø),"X" ELSE !TAB(2Ø),"Y"
```

5.5.5 Call Machine Language (CALL)

```
     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 1ØØ 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 Ø 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)
```

5.5.6 Address of a Variable (ADDR)

```
     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)
```

5.5.7 CPMFN Call Function (CP/M Version Only)

CPMFN allows for execution of any of the CP/M System Function Calls directly from baZic and may be issued directly as a command or executed as a statement. The syntax follows the function syntax.

The function is called by passing the function number as the first argument to the function call. 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, this value in the CPMFN call must be included.

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

An example is as follows:

```
     CPMFN (<C ARGUMENT>)[,<DE ARGUMENT>]
```

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.

6.1 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	5Ø=5*1Ø
/	DIVISION	1Ø=5Ø/5
-	SUBTRACTION	12=25-13
+	ADDITION	15=5+1Ø
-	NEGATION	-5=-(+5)

6.2 Relational Operators

The relational operators are the true/false operators. The result of a relational comparison is either true (=1) or false (=Ø). 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 (Ø)
=	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

6.3 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:

```
1Ø X=1
2Ø Y=5
3Ø IF X=1 AND Y=5 THEN (THE EXPRESSION IS TRUE)
4Ø 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.

```
1Ø X=1
2Ø Y=5
3Ø IF X=1 OR Y=5 THEN (THE EXPRESSION IS TRUE)
4Ø IF X=1 OR Y=1 THEN (THE EXPRESSION IS TRUE)
5Ø IF X=Ø OR Y=5 THEN (THE EXPRESSION IS TRUE)
6Ø IF X=Ø 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.

```
10 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 saves three bytes of program storage.

6.4 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. Parenthesis 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 an operator, but evaluated first)
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 quickly 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 can fit on one line 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
```

7.1 Define a Function (DEF)

```
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.

7.2 Return from a Function (RETURN)

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

The RETURN statement is used to terminate a multi-line 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 4.3.7 for more information.

Examples of the use of the RETURN statement follow:

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

7.3 Function End (FNEND)

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 7 (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.

8.1 Trappable Errors

Trappable errors are errors which can be trapped using the ERRSET statement of **baZic**. (See Section 8.1) 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.

8.1.1 ARGument (Error 1)

The ARGument error is returned any time 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.

8.1.2 DIMENSION (Error 2)

The DIMENSION error refers to a problem using the DIM statement. Either the programmer has tried to redimension 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.

8.1.3 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 Ø 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 1Ø. 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.Ø62
LOG	Zero argument
	Argument less than 3.1622776E-64.
	Argument less than zero.
SETPO	Any attempt to set print device greater than 7.

8.1.4 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 nibble of BCD number is zero.
FOR	Index variable is a string variable.

8.1.5 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.

8.1.6 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.

8.1.7 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.

8.1.8 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. Micro Mike's, Inc., 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    WRITE    OPEN      READ
```

8.1.9 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)

8.1.10 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 ØDH (carriage return) before a double quote.
EXCFN	Missing right parenthesis.
$CONST	String constant. Finds ØDH before a double quote.
VFYB	Verify that next token is equivalent to value in Register B. Most frequent use is to verify expected left and right parenthesis and commas.
CHEKEY1	A routine which checks end of statement. Gives error on improper ending.
LVR	Expecting a variable but didn't find one.
CHK$	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.

8.1.11 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.

8.1.12 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.

8.1.13 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.

8.1.14 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 (Ø). 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.

8.1.15 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.

8.1.16 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.

8.2 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.

8.2.1 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, or the Control C or a STOP was executed. For this reason these errors cannot be trapped using the ERRSET command.

8.2.2 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.

8.2.3 DOUBLE DEFinition

The DOUBLE DEFinition error always occurs at RUN time when all user defined functions are evaluated. This error is returned when two or more user defined functions are found to have the same name.

8.2.4 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.

8.2.5 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 3 (DIRECT COMMANDS) for a list of statements that can be used in the direct mode.

8.2.6 INTERNAL STACK OVerflow

This error is not normal and should not occur under normal use. If this error does occur, contact Micro Mike's, Inc. A copy of all disks and a description of the circumstances may be required to duplicate the problem.

8.2.7 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.

8.2.8 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.

8.2.9 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.

8.2.1Ø 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 consider the internal line editor of **baZic**. The following sections will cover how **baZic** stores information in disk files and the major differences between **baZic** and other dialects of BASIC.

9.1 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, **baZic** has 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 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 each 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** displays the line number requested automatically. 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 letter is pressed.)

9.1.1 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 three 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, or add to 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.

9.1.2 Control N

The Control N command is discussed partially 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.

9.1.3 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.

9.1.4 Control Q

The backspace command, Control Q 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.

9.1.5 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. 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.

9.1.6 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.

9.1.7 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.

9.2 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, CP/M or NorthStar DOS) is responsible for keeping track of all files, and the accessing of these files is done from **baZic** through the Disk Operating System. See your operating system 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.

The maximum size of any one file is 16.7 million characters (16,776,960 bytes) under MicroDoz and 4 megabytes under CP/M, 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.

MicroDoZ does not support dynamic file allocation. (i.e. MicroDoZ files cannot expand automatically when the file size is exceeded.) This means that all files should be the proper size or greater when created. CP/M files are Dynamic

Before any file can be used by **baZic**, it must be OPENed. See Section 4.4.3 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 more than 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
1Ø	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, Ø=positive) and the remaining bits (6-Ø) 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 Ø).

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.

9.3 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.

9.3.1 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 more convenient method of defining substrings within a larger string.

The position of the substring is passed to any string argument exactly the same 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.

9.3.2 IF THEN Evaluation

Many BASICs evaluate the IF THEN condition differently than **baZic.** In **baZic,** should the IF THEN evaluation be 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 the same way 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
```

9.3.3 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 1Ø under baZic (B(1Ø)), the array actually has 11 elements (Ø to 1Ø). If memory is at a premium in your system and you only need 1Ø 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 mathematics 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.

10.1 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     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 3Ø. 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 2DØØH 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 MicroDoZ by typing BYE and save **baZic** on the disk by typing SF BAZIC Ø1ØØ (MicroDoZ version.) The procedure would appear as follows:

```
     READY
     BYE
     1>SF BAZIC 1ØØ
     1>JP 1ØØ
     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

	baZicØ8	baZic1Ø	baZic12	baZic14
ATN	166	177	188	199
COS-SIN	395	432	454	476
LOG	657	7Ø4	75Ø	784
EXP	957	1Ø15	1Ø79	1125
^	1Ø25	1Ø83	1147	1193
SQRT	1174	1234	13ØØ	1348
RND	1266	1326	1392	144Ø

10.2 XREF

The XREF program is not available with 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 debugging 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      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. Following is a sample listing, which in turn is followed by a sample XREF table which shows the number of symbols used by the program and the total number of times all symbols are used. On successful completion of the listing, or if an error is encountered, XREF returns to the XX4H (ORIGIN+4) 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 BDØØH 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 BDØØH. 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 XREF.

This program will work with all eight versions of **baZic**, i.e., 8, 1Ø, 12, and 14 digit precisions in both the hardware floating point and software floating point versions.

Sample EXREF Listing

```
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 4 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 routines is",E1-R," bytes."
290 !#P\!#P
300 END
```

Sample EXREF Table

A	80,180,190
A$	40,60,120,160,220
C	90,190,200
E	110,210,230,250
E1	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

10.3 Interfacing COPY Programs to baZic Programs

(for MicroDoZ version only)

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). See the MicroDoZ manual for a description of these programs.

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.

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.

This page left blank intentionally.

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.

12.1 MicroDoZbaZic Partial Source Listing

```
     Ø1ØØ          ORG ORIGIN
Ø1ØØ AF           ENTRY1 XRA A              Clears program
Ø1Ø1 18 Ø2         JR RENTR1
Ø1Ø3 Ø1           MEMFLG DB 1               Auto MEMSET if 1
Ø1Ø4 37           ENTRY2 STC                Saves program
Ø1Ø5 21 32AA      RENTR1 LXI H,ENDBAS
Ø1Ø8 11 8ØØØ       LXI D,Ø8ØØØH             End memory
Ø1ØB C3 Ø138       JMP INTL1
                  ***
Ø1ØE 5Ø           LINECT DB 5ØH             Initial line length
Ø1ØF Ø1           AUTOS DB 1                Auto-start flag
Ø11Ø E8           PROM DB ØE8H              Disk controller origin
Ø111 282Ø         LINETB DW PHPOS           Print head table
Ø113 18           PAGES DB 18H              CRT Lines per page
                  *
Ø114 C3 Ø231      ENTRY3 JMP READY1         Saves data
                  *
Ø117 Ø8           BSPC DB 8H                Character delete
Ø118 ØØ           CONC DB Ø                 Control C inhibit flag

Ø122 ØØØ3          DS 3
Ø125 FB           ENABLE EI
Ø126 C9            RET
Ø127 ØØ            NOP
                  ;
Ø128 31F9         SBDATA DW ATN             Addresses used by SHORTB
Ø12A 3ØFA          DW COS                   program.
Ø12C 2FEA          DW LOG
Ø12E 2EB3          DW EXP
Ø13Ø 2E6F          DW TOPWR
Ø132 2DD8          DW SQRT
Ø134 2D8C          DW RND1
Ø136 2EB1          DW TOPWR4+1
```

Control C can be inhibited by FILLing the byte at 28Ø Decimal with a byte value of 1. Control C can be re-enabled by FILLing this byte back to zero (Ø).

This page left blank intentionally.

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.

12.1 MicroDoZbaZic Partial Source Listing

```
     0100          ORG ORIGIN
0100 AF           ENTRY1 XRA A           Clears program
0101 18 02         JR RENTR1
0103 01           MEMFLG DB 1            Auto MEMSET if 1
0104 37           ENTRY2 STC             Saves program
0105 21 32AA      RENTR1 LXI H,ENDBAS
0108 11 8000       LXI D,08000H          End memory
010B C3 0138       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 2820         LINETB DW PHPOS        Print head table
0113 18           PAGES DB 18H           CRT Lines per page
                  *
0114 C3 0231      ENTRY3 JMP READY1      Saves data
                  *
0117 08           BSPC DB 8H             Character delete
0118 00           CONC DB 0              Control C inhibit flag

0122 0003          DS 3
0125 FB           ENABLE EI
0126 C9            RET
0127 00            NOP
                  ;
0128 31F9         SBDATA DW ATN          Addresses used by SHORTB
012A 30FA          DW COS                program.
012C 2FEA          DW LOG
012E 2EB3          DW EXP
0130 2E6F          DW TOPWR
0132 2DD8          DW SQRT
0134 2D8C          DW RND1
0136 2EB1          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 bootup by setting the AUTOS byte (010FH) to 0 (FILL 201,0) 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.

12.2 baZic for CP/M Partial Source Listing

CPM BAZIC

```
     0100        ORG ORIGIN
0100 AF         ENTRY1 XRA A          Clears program
0101 18 02       JR RENTR1
0103 01         MEMFLG DB 1           Auto MEMSET if 1
0104 37         ENTRY2 STC            Saves program
0105 21 355F    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         PRCM DB 0E8H          Disk controller origin
0111 29DD       LINETB DW PHPCS       Print head table
0113 18         PAGES DB 18H          CRT Lines per page
                *
0114 C3 0267    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 WH19 CRT. On entry register A
                ;      contains the device number, register D the
                ;      column and register E the line.
0119 FF         GOTOYX DB 0FFH        Indicates assembly language routine
011A D5          PUSH D               follows.
011B 0E 06       MVI C,6
011D 1E 1B       MVI E,27
011F CD 2934     CALL BDOSC           Use this address for BDOS calls to save
0122 0E 06       MVI C,6              IX & IY if you have a CP/M look-a-like
0124 1E 59       MVI E,"Y"            that uses the IX & IY registers.
0126 CD 2934     CALL BDOSC
```

```
Ø129 2A Ø144      LHLD OFFSET
Ø12C D1           POP D
Ø12D 19           DAD D
Ø12E E5           PUSH H
Ø12F ØE Ø6        MVI C,6
Ø131 5D           MOV E,L
Ø132 CD 2934      CALL BDOSC
Ø135 E1           POP H
Ø136 ØE Ø6        MVI C,6
Ø138 5C           MOV E,H
Ø139 CD 2934      CALL BDOSC
Ø13C C9           RET
                  ENDIF
Ø13D ØØØ7         DS 7
Ø144 1F1F         OFFSET DW 1F1FH       Zenith WH19 offset
                  ;       Clear screen routine. The example below is for the
                  ;       Zenith WH19 CRT. On entry register A contains the
                  ;       device number.
Ø146 FF           CLS1 DB ØFFH
Ø147 ØE Ø6         MVI C,6              Direct console output
Ø149 1E 1B         MVI E,27             Zenith WH19 clear screen sequence
Ø14B CD 2934       CALL BDOSC           is ESC "E"
Ø14E ØE Ø6         MVI C,6
Ø15Ø 1E 45         MVI E,"E"
Ø152 CD 2934       CALL BDOSC
Ø155 C9            RET
Ø156 ØØØ3          DS 3
Ø159 34AE         SBDATA DW ATN         Addresses used by SHORTB
Ø15B 33AF          DW COS               program.
Ø15D 329F          DW LOG
Ø15F 3168          DW EXP
Ø161 3124          DW TOPWR
Ø163 3Ø8D          DW SQRT
Ø165 3Ø41          DW RND1
Ø167 3166          DW TOPWR4+1
```

Control C can be inhibited by FILLing the byte at Ø118H (Decimal 28Ø) with a byte value of 1. Control C can be re-enabled by FILLing this byte back to zero (Ø).

A panic stop, normally a Control C, can be changed to another control character by modifying the following locations within baZic for CP/M (Release Ø5/Ø4): Ø336H, 1CØFH and 1F28H. A 3H (^C) value will be found at these locations and should be changed to the value needed.

Other locations within baZic for CP/M (Ø5/Ø4) which may be needed include:

Control A	Ø377H	Control Y	Ø398H
Control G	Ø37BH	Control D	Ø395H
Control H	Ø383H	Control N	Ø39EH
Control Z	Ø387H	Control Q	Ø37FH

baZic can be made to RUN a program automatically upon bootup by setting the AUTOS byte (Ø1ØFH) to Ø (FILL 271,Ø) 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 (Ø) 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 **baZic** with the name of the program to be executed, e.g.:

```
A>BAZIC1Ø PROGRAM
```

ASCII TABLE

ASCII	DECIMAL	HEX	OCTAL	ASCII	DECIMAL	HEX	OCTAL
NUL	Ø	ØØ	ØØØ	SP	32	2Ø	Ø4Ø
SOH	1	Ø1	ØØ1	!	33	21	Ø41
STX	2	Ø2	ØØ2	"	34	22	Ø42
ETX	3	Ø3	ØØ3	#	35	23	Ø43
EOT	4	Ø4	ØØ4	$	36	24	Ø44
ENQ	5	Ø5	ØØ5	%	37	25	Ø45
ACK	6	Ø6	ØØ6	&	38	26	Ø46
BEL	7	Ø7	ØØ7	'	39	27	Ø47
BS	8	Ø8	Ø1Ø	(	4Ø	28	Ø5Ø
HT	9	Ø9	Ø11	)	41	29	Ø51
LF	1Ø	ØA	Ø12	*	42	2A	Ø52
VT	11	ØB	Ø13	+	43	2B	Ø53
FF	12	ØC	Ø14	,	44	2C	Ø54
CR	13	ØD	Ø15	-	45	2D	Ø55
SO	14	ØE	Ø16	.	46	2E	Ø56
SI	15	ØF	Ø17	/	47	2F	Ø57
DLE	16	1Ø	Ø2Ø	Ø	48	3Ø	Ø6Ø
DC1	17	11	Ø21	1	49	31	Ø61
DC2	18	12	Ø22	2	5Ø	32	Ø62
DC3	19	13	Ø23	3	51	33	Ø63
DC4	2Ø	14	Ø24	4	52	34	Ø64
NAK	21	15	Ø25	5	53	35	Ø65
SYN	22	16	Ø26	6	54	36	Ø66
ETB	23	17	Ø27	7	55	37	Ø67
CAN	24	18	Ø3Ø	8	56	38	Ø7Ø
EM	25	19	Ø31	9	57	39	Ø71
SUB	26	1A	Ø32	:	58	3A	Ø72
ESC	27	1B	Ø33	;	59	3B	Ø73
FS	28	1C	Ø34	<	6Ø	3C	Ø74
GS	29	1D	Ø35	=	61	3D	Ø75
RS	3Ø	1E	Ø36	>	62	3E	Ø76
US	31	1F	Ø37	?	63	3F	Ø77

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	8Ø	2ØØ	16Ø	AO	24Ø
129	81	2Ø1	161	A1	241
13Ø	82	2Ø2	162	A2	242
131	83	2Ø3	163	A3	243
132	84	2Ø4	164	A4	244
133	85	2Ø5	165	A5	245
134	86	2Ø6	166	A6	246
135	87	2Ø7	167	A7	247
136	88	21Ø	168	A8	25Ø
137	89	211	169	A9	251
138	8A	212	17Ø	AA	252
139	8B	213	171	AB	253
14Ø	8C	214	172	AC	254
141	8D	215	173	AD	255
142	8E	216	174	AE	256
143	8F	217	175	AF	257
144	9Ø	22Ø	176	BØ	26Ø
145	91	221	177	B1	261
146	92	222	178	B2	262
147	93	223	179	B3	263
148	94	224	18Ø	B4	264
149	95	225	181	B5	265
15Ø	96	226	182	B6	266
151	97	227	183	B7	267
152	98	23Ø	184	B8	27Ø
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	19Ø	BE	276
159	9F	237	191	BF	277

ASCII TABLE

DECIMAL	HEX	OCTAL	DECIMAL	HEX	OCTAL
192	CØ	3ØØ	224	EØ	34Ø
193	C1	3Ø1	225	E1	341
194	C2	3Ø2	226	E2	342
195	C3	3Ø3	227	E3	343
196	C4	3Ø4	228	E4	344
197	C5	3Ø5	229	E5	345
198	C6	3Ø6	23Ø	E6	346
199	C7	3Ø7	231	E7	347
2ØØ	C8	31Ø	232	E8	35Ø
2Ø1	C9	311	233	E9	351
2Ø2	CA	312	234	EA	352
2Ø3	CB	313	235	EB	353
2Ø4	CC	314	236	EC	354
2Ø5	CD	315	237	ED	355
2Ø6	CE	316	238	EE	356
2Ø7	CF	317	239	EF	357
2Ø8	DØ	32Ø	24Ø	FØ	36Ø
2Ø9	D1	321	241	F1	361
21Ø	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	33Ø	248	F8	37Ø
217	D9	331	249	F9	371
218	DA	332	25Ø	FA	372
219	DB	333	251	FB	373
22Ø	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

High Nibble

Low	8	9	A	B	C	D	E	F
Ø	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	CPMFN	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 version only.
CPMFN available in CP/M version only.

INDEX