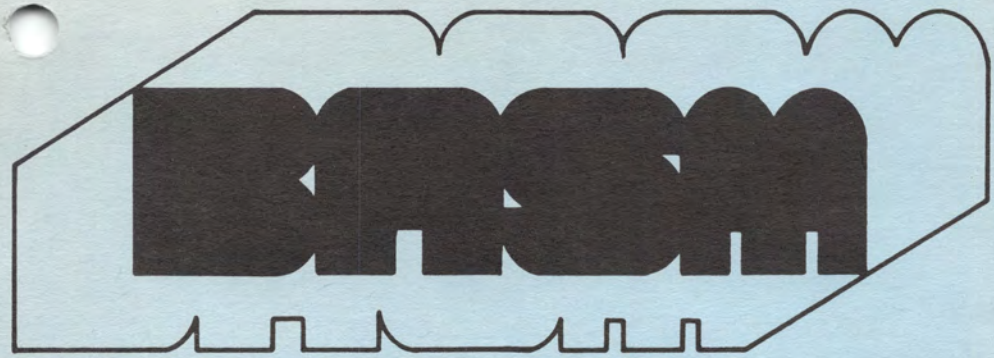

BASIC COMPILER AND ASSEMBLER
for the ATARI® 400/800 Computer

Published and Distributed by

COMPUTER ALLIANCE

21115 Devonshire • Chatsworth, Ca. 91311 • Suite 132

CONTENTS

OVERVIEW OF BASM

WHAT IS BASM?

This is the BASM Basic Compiler and Assembler for the Atari 400/800 computer.

Many programs written in BASIC also have portions that are written in ASSEMBLY LANGUAGE, because of a need for faster program execution. BASM is a BASIC that thinks like an ASSEMBLY LANGUAGE, because it uses the syntax of BASIC with ASSEMBLY LANGUAGE data types and addressing modes. The BASM BASIC Compiler and Assembler takes what you program in BASIC and compiles it into a binary code that talks directly to the machine. Thus, it is the next step in the evolution of the small computer BASIC language.

BASM has features for both the BASIC programmer and the programmer who is already familiar with ASSEMBLY LANGUAGE.

BASM provides the BASIC programmer with a familiar language with which you can produce a program that requires a much faster speed, such as good graphic animation. BASM produces programs that run as much as 130 times faster than Atari BASIC. Also, if you are a programmer currently using BASIC and you desire to learn a machine language, BASM becomes an excellent means by which you can ease your way into learning ASSEMBLY LANGUAGE programming.

NOTE: For the user who is mostly familiar with BASIC, it should be noted here that BASM is not created to be directly compatible with Atari BASIC; for example,

you cannot expect to run programs that have been previously written in Atari BASIC through this compiler. The commands are not always the same as Atari BASIC, and vice versa. However, many of the commands are similar, and all are easy to understand.

For the ASSEMBLY LANGUAGE programmer, BASM retains the efficiency of ASSEMBLY, yet enables you to cut down the time required for program development by as much as two to three times, because it eliminates the tedium of calculating the logistics of ASSEMBLY LANGUAGE syntax. Therefore, it produces programs that document better and are easier to understand. BASM itself also makes a good assembler if you do not wish to use BASIC.

However, it is certain that once you have become familiar with the BASM BASIC Compiler, you will discover that it is a valuable programming tool that gives you a lot of control over what you are writing, and it will serve to optimize your program development time.

HOW TO RUN BASM

To use the BASM BASIC Compiler and Assembler, the following equipment is needed:

Atari 400 or 800 system with
32K or more of memory
Atari 810 Disk System

Optional:
Atari-compatible printer
Second Atari disk drive

It is recommended that you make a back-up disk as soon as possible. We have used quality disks in the production of BASM, but it is always good to have a backup.

To run BASM, power up, using the BASM disk (the Atari BASIC cartridge should not be inserted). Type L (return) BASM (return). This will load in the BASM program. BASM begins in the EDITOR mode which is also the command mode.

EDITOR COMMANDS

The BASM editor commands act like the editing commands of Atari BASIC.

Each line has a line number. Line numbers may range from 1 to 65534 inclusive. You may insert a line by typing the number followed by the line which you wish to insert. To erase a line, you type the line number without the line following it. Note that the line numbers in BASM are temporary and for editing purposes only. They may not be used as label references in **GOTO** statements or **GOSUB** statements. The actual lines start in the first column following the last digit in the line number. The file name references are the standard ATARI file names, but if they are on Disk D1:, you may omit the prerequisite D: or D1:.

The following commands are available on the BASM editor. Unless otherwise noted, the parameters may follow immediately after the command, or may be separated by space(s).

COMPILE: The short form of **COMPILE** is the letter "C". Follow this command by a carriage return. This invokes the BASM compiler to process the source text file that you created using the editor, to produce a binary file which is actually runable. This command first asks you the SOURCE FILE name. After you type in the SOURCE FILE name, it asks for the BINARY FILE name. After you type that in, it asks for the LIST FILE name. Typing **P:** in response to LIST FILE name will send the listing to the printer. Typing **E:** will send the listing to the screen. Typing a RETURN by itself will prevent any LIST FILE from being created. This represents a

significant speed improvement in compilation. Example:

```
C
SOURCE FILE NAME ->TEST.SRC
BINARY FILE NAME ->TEST
LIST   FILE NAME ->P:
```

DELETE: The short form of **DELETE** is the letters "DE". Follow this by two line numbers, separated by a comma. This will delete all lines between, and including, the two line numbers you gave. Example:

```
DE100,130
```

DOS: The short form is "DO". This returns you to the Atari Disk Operating System, and exits BASM entirely. Example:

```
DO
```

FIND: The short form is "F". Follow this by a "/", then print the text you wish to find, followed by another "/". In order to allow you to find text in which the "/" is included, you may substitute any character for the "/" as delimiters for your text. **FIND** will then search through your entire program, find the first occurrence of your text, stop, and print out that line. You may then go on to find the next occurrence of that same text by typing **FIND**, followed directly by a RETURN. This process may be repeated over and over as you wish, until you come to the end of your program. Also, **FIND** will not be confused by any changes you make to your program during this process. Examples:

```
F/DEF PRHEX/
```

```
F"YES/NO"
FIND  /2/
```

FREE: The short form is "FR". This tells you how many free bytes are left for the editor to use. Example:

```
FR
```

INSERT: The short form is "I". Follow this by two numbers, separated by a comma. Example: I1ØØ,5. In this example, five line numbers would be inserted in front of line number 1ØØ. This command does not actually insert blank lines, but renumbers the lines that follow. This creates a gap in the line number editing system to allow you to insert new program lines. If the comma and second number are omitted, it defaults to inserting 1ØØ lines. Examples:

```
I1ØØ,5
I 1ØØ,5
I935
```

LIST: The short form is "L". The **LIST** command, followed by a single line number, will print that line, and that line only, on the screen. The **LIST** command, followed by two line numbers separated by a comma, will list those two lines, and all the lines between them, in order. The **LIST** command, followed by a comma and then a line number, will list all lines from the beginning of the program up to and including that number. The **LIST** command, followed by a line number and then a comma, will list all the lines to the end of your program, starting at and including that number. The **LIST** command followed by a return will list the entire program. The program listing may be stopped

with the BREAK key. Examples:

```
L
L1Ø
L1Ø,4Ø
L ,4Ø
L 4Ø,
LIST
```

LOAD: The short form is "LO". Follow this command by the file name of your source program. This loads your program into the text editor buffer for editing. It also automatically assigns line numbers to your program by tens, starting with 1Ø. (There must be at least one space before the file name.) Examples:

```
LO  TEST.SRC
LOAD TEST.1
```

NEW: The short form is "N". This command erases your program from the editor buffer. Example:

```
NEW
```

RENUMBER: The short form is "RE". This command, followed by a RETURN, will renumber your program by tens, starting at 1Ø. If the **RENUMBER** command is followed by a number, it will renumber your program in intervals of that number, starting at that number. Examples:

```
RE
REN
RENU
RENUMBER
RE1Ø
REN 1ØØ
```

RE17

RUN: The short form is "R". Follow this command by the file name of the binary file you wish to run. (There must be at least one space before the file name.) Examples:

```
R TEST
RUN TEST
```

SAVE: The short form is "S". This command saves the contents of the editor buffer to disk. **Note that the line numbers are discarded at this point.** The **SAVE** command is followed by the file name of your source program. (There must be at least one space before the file name.) Examples:

```
S TEST.SRC
SAVE  TEST.SRC
```

EXAMPLES

BEGINNING BASM

1. Copy your disk

Copy your entire distribution disk to a blank floppy, using the 'J', copy entire disk command. You will need some files on the disk besides just BASM.

2. Boot off of the disk you have just created, without the BASIC cartridge.

3. Use the DOS command 'L' to load BASM (type L[return] BASM[return]).

4. BASM should now be in the edit/command mode. It should have identified itself and printed READY.

MAZE

Type in the following exactly. Be sure to type the indicated spaces, and don't insert any extra spaces. Also, don't confuse Ø's with O's.

```
1ØCONSOL=$DØ1F
2ØRANDOM=$D2ØA
3Ø WHILE CONSOL AND 1 <> Ø
4Ø  PUT RANDOM AND 1 + 6
5Ø ENDWHILE
6Ø RETURN
```

You have now entered a short program into the editor workspace memory. You may view the contents of the workspace memory by

typing L[return]. In order to run your program, you must first save it to disk, and compile it. The following will save your program to disk in a file named MAZE.SRC .

```
SAVE MAZE.SRC
```

You may now compile your program. After you type COMPILE[return], BASM will ask you for the name of the source file. This means the file that you just saved to disk (what you actually type in). It then asks you for the binary file name. This is the final program in machine code that BASM creates during compilation. Finally, BASM asks you for the list file name. This is a documentation file useful in de-bugging. You may skip the creation of this file by just typing [return] with no file name. The following will take the program you just saved to disk, compile it, save the binary program in a file named MAZE, and produce no list.

```
COMPILE
SOURCE FILE NAME -->MAZE.SRC
BINARY FILE NAME -->MAZE
LIST   FILE NAME -->
```

You may now run your program by typing the following.

```
RUN MAZE
```

You can stop your program and return to BASM by pressing the [START] key.

HOW IT WORKS

Load your program back into workspace memory and display it by typing the following.

```
LOAD MAZE.SRC
LIST
```

Lines 1Ø and 2Ø allow you to define Atari hardware control registers as variables. The $DØ1F does not put that value into CONSOL; it locates the variable CONSOL at that location in "memory." These lines start right after the line number because they are actually assembly language and not BASIC; this is the format for standard 65Ø2 assembly.

Line 3Ø is the start of a WHILE loop. This is similar to a FOR/NEXT loop, except that it loops while a condition is true, rather than counting. In this case, it reads the Atari CONSOL register, masks off all bits of it except bit Ø, and loops while this bit is not equal to zero. This bit becomes zero when the [START] key is pressed.

Line 4Ø reads the random number register, masks off all bits except bit Ø and adds 6 to it. It then sends the result to the screen. This will randomly print left and right diagonal lines. This line has an extra space after the line number. The space does not affect the program at all, but is optional documentation to indicate that this line is inside a loop. Note that BASM expressions are always evaluated from left to right.

Line 5Ø is the bottom of the WHILE loop.

Line 60 returns you to BASM. Running your program from BASM is exactly the same as doing a GOSUB to the very top of your program. You can also run your program from Atari DOS by using the L command from DOS, and the RETURN on line 60 will return you to DOS.

SEEING THE ASSEMBLY

Now type the following line (your program should still be in workspace memory).

```
25 .LST CODE
```

This tells BASM to show you the assembly code produced by your BASM program. Using the SAVE and COMPILE commands, save and compile your program, except that when it asks you for the list file name, type E:[return]. This will clear the screen and then send the listing to the screen. What you will see is your program with assembly language intermixed.

A BIGGER PROGRAM

This program is a little larger and a lot more useful. It allows us to see the disk directory from BASM. Type in the following exactly:

```
1Ø TRAP .NOTHING
2Ø PRINT "FILE NAME?";
3Ø INPUT LINE$
4Ø OPEN 1 , 6 , Ø , LINE$
5ØLOOP FILE 1
6Ø INPUT LINE$
7Ø IF STATUS < 128 THEN
8Ø  FILE Ø : PRINT LINE$
9Ø  GOTO LOOP
1ØØ ENDIF
11Ø CLOSE 1 : FILE Ø
12Ø RETURN
13Ø;
14ØNOTHING RETURN
15Ø DIM LINE$(1ØØ)
```

Save it to disk with the filename CAT.SRC Compile it using the C command (BASM editing and control commands have short forms: C is COMPILE). Give the binary file the name CAT. Now run the program CAT. It should ask you FILE NAME? Type *.* and you should see the entire directory of the disk.

HOW IT WORKS

Line 1Ø causes the subroutine "NOTHING" (line 14Ø) to execute whenever an error is detected. The period in front of NOTHING

tells BASM to give the location of NOTHING to the TRAP program, rather than the data at NOTHING. If you don't use this feature, then any I/O errors will re-boot you back into Atari DOS, including end of file error.

Line 2Ø prints the words FILE NAME? on the screen without a [return]. The PRINT program only works for string constants and variables; BPRINT is for numeric variables.

Line 3Ø inputs a line from the keyboard and saves it in memory at LINE$.

Line 4Ø opens the disk directory and gives it the file number, 1

Line 5Ø switches input/output to file number 1. In BASM, you may open a number of files and switch between them, using the FILE statement. This process does not either open or close the files, but replaces the Atari #number in the INPUT, PRINT, etc. statements. File Ø is automatically opened to device E: (screen/keyboard). This line also defines a location which can be referred to at other places in the program, such as at line 9Ø, and replaces the Atari line number mechanism. BASM line numbers are for editing purposes only, and are not even saved on disk.

Line 6Ø inputs one line of directory from the disk.

Line 7Ø checks for any error conditions (in this case the end of file). Since nothing follows the THEN, this is a multiple-line IF/THEN statement. Lines 8Ø & 9Ø are executed if line 7Ø is true; otherwise it skips down to line 11Ø.

Line 8Ø switches to the screen and prints out what you just got from the disk

directory.

Line 90 goes back to LOOP to get another line of the directory. In BASM it is legal and useful to jump out of IF/ENDIF's, WHILE/ENDWHILE's and FOR/NEXT's.

Line 100 is the bottom of the IF statement.

Line 110 closes the disk directory and switches back to screen/keyboard mode.

Line 120 returns to BASM.

Line 130 is the assembly language equivalent of the BASIC REM statement.

Line 140 is a "do-nothing" routine that is referenced in line 10.

Line 150 creates "LINE$". Note that this line is never actually executed, but it exists in the program. You could put it anywhere you want, but if your program tries to execute it, it will probably "BOMB OUT."

FROM ATARI BASIC

The first program, program A, is in Atari BASIC. It inputs your name, and prints it backwards. This is not directly compatible to BASM, but the program B is in BASM and performs the same action.

Program A: Atari BASIC

```
10 DIM NAME$(100)
20 PRINT "YOUR NAME";;
30 INPUT NAME$
40 N=LEN(NAME$)
50 PRINT "BACKWARDS: ";
60 PRINT NAME$(N,N);
70 N=N-1
80 IF N<>0 THEN GOTO 60
90 PRINT
```

Program B: BASM

```
10 PRINT "YOUR NAME?";
20 INPUT NAME$
30 LDY #0
40 WHILE NAME,Y <> #EOL
50   INY
60 ENDWHILE
70 STY N
80 PRINT "BACKWARDS: ";
90 WHILE N <> 0
100   LDY N : DEY : PUT NAME,Y
110   DEC N
120 ENDWHILE
130 PUT #EOL
140 RETURN
150 DIM NAME$(100) , N
```

Save the BASM program B to disk, using file name REV.SRC, compile it using binary

file name REV, and run it using file name REV.

HOW IT WORKS

Line 1Ø of the program B is the equivalent of line 2Ø in program A. The program B includes a question mark, because the BASM INPUT command (line 2Ø) does not print a question mark as in Atari BASIC. The semicolon prevents the cursor from going to the next line. Only one variable or constant is allowed in the PRINT statement.

Line 2Ø of program B INPUTS a line from the keyboard to memory, starting at NAME$. It stores whatever you type in, including the [return] character, and then adds a binary Ø after the [return]. After being loaded into memory, the data can be treated as a byte "array"; BASM uses 65Ø2 indexed addressing modes rather than classical array processing, to access this kind of data.

Lines 3Ø through 7Ø of program B are the rough equivalent of line 4Ø of the Atari BASIC program A.

Line 3Ø loads Ø into the 65Ø2 Y register. The Y register is used to index into the NAME$ "array" in line 4Ø, and Ø points to the first byte of that "array."

Line 4Ø loops if the byte indexed by the Y register is not equal to [return]. The #EOL represents the end of line or [return] character. The # means to treat EOL as a constant number rather than a variable. This is a convenient way of defining special characters and values. EOL was defined in the library system by an EOL=$9B statement. The "NAME,Y" is the 65Ø2 indexed addressing, and it means "get one byte from memory yy

places beyond the location in memory called NAME, where yy is the contents of the Y register." Note that the $ of NAME$ is dropped, because we are now treating NAME$ as the byte "array" NAME. NAME and NAME$ refer to the same thing but treat it differently.

Line 5Ø adds 1 to the Y register, which is used to access the next character in NAME.

Line 6Ø is the bottom of the WHILE/ENDWHILE loop, and loops back up to line 4Ø.

Line 7Ø stores the contents of the Y register to the variable N. At this point, Y indexes to the EOL character in NAME. This is also the length of NAME$ minus the EOL.

Line 8Ø prints the label "BACKWARDS: "; and works the same as line 5Ø in program A.

Line 9Ø is another loop, this time checking to see if the variable N indexes to the beginning of NAME$. It stops looping when it is done.

Line 1ØØ loads the index N into the Y register, subtracts 1 from the Y register, indexes into NAME, and sends the indexed byte from NAME$ to the screen, using the PUT statement. It subtracts 1 because BASM indexes, starting at Ø, where as Atari BASIC indexes, starting at 1. You have to load the Y register each time in the loop, because the PUT program alters all the 65Ø2 registers. It is good to assume that any library commands alter all of the 65Ø2 registers. Chapter 5 of the manual tells you if a command is library or built-in.

Line 11Ø subtracts 1 from the index N. You could have used "LET N = N - 1", but "DEC N"

executes faster and takes up less memory.

Line 12Ø is the bottom of the WHILE/ENDWHILE loop started in line 9Ø.

Line 13Ø ouputs an EOL to the screen.

Line 14Ø returns to BASM or DOS, from wherever this program was called. BASM programs should always finish execution with a RETURN (or STOP); otherwise, they will continue executing whatever comes next in memory (which is not a good idea).

Line 15Ø creates the variables NAME$ and N. All variables in BASM must be explicitly created, either by the DIM statement, or by assembly language means.

SOME GRAPHICS

The previous examples have not shown the real advantage: **SPEED!** This program shows you how to use graphics and how much faster BASM is than Atari BASIC.

Program C: Atari BASIC

```
1Ø GRAPHICS 7+16
2Ø X=4Ø:Y=4Ø
3Ø COLOR RND(Ø)*4
4Ø X=X+INT(RND(Ø)*3-1)
5Ø IF X=16Ø THEN X=159
6Ø IF X=-1 THEN X=Ø
7Ø Y=Y+INT(RND(Ø)*3-1)
8Ø IF Y=96 THEN Y=95
9Ø IF Y=-1 THEN Y=Ø
1ØØ PLOT X,Y
11Ø IF PEEK(53279)<>16 THEN 3Ø
```

Program D: BASM

```
1ØRANDOM=$D2ØA
2ØCONSOL=$DØ1F
3Ø GR7
4Ø LET 4Ø -> X -> Y
5Ø WHILE CONSOL AND 1 <> Ø
6Ø  COLOR RANDOM AND 3
7Ø  RND 2 , .TMP
8Ø  LET X = X + TMP - 1
9Ø  IF X = 16Ø THEN LET X = 159
1ØØ  IF X = 255 THEN LET X = Ø
11Ø  RND 2 , .TMP
12Ø  LET Y = Y + TMP - 1
13Ø  IF Y = 96 THEN LET Y = 95
14Ø  IF Y = 255 THEN LET Y = Ø
15Ø  PLOT7 X , Y
16Ø ENDWHILE
17Ø CLOSE 7
```

```
18Ø FILE Ø
19Ø RETURN
2ØØ DIM X , Y , TMP
```

Enter and save the second program to a file named SCRIBBLE.SRC. This program uses the library routines GR7, PLOT7, and RND, which might not be included in the library system. These routines are in a file named MISC. To check, load the file named BASM.LIB into the editor workspace (LOAD BASM.LIB[return]). It should have a line: .INCL 'MISC' . If it doesn't, add .INCL 'MISC' on the line following the line: .INCL 'GR' . You can insert lines in the same way as you do in Atari BASIC. Save it back to the file BASM.LIB. This causes the file MISC to be included in your program when you compile. You may now compile and run your program SCRIBBLE.

HOW IT WORKS

Lines 1Ø and 2Ø define the locations of CONSOL and RANDOM.

Line 3Ø opens the graphics mode 7 with no text. It uses file number 7 for INPUT/OUTPUT.

Line 4Ø sets up the initial location on the screen. This form of the LET uses the assign (->) rather than equal (=) and allows you to store the same value in two or more variables.

Line 5Ø loops to line 16Ø until the [START] key is pressed.

Line 6Ø selects a random color between Ø and 3. It masks off the lower 2 bits from the random number generator and gives them to the COLOR program.

Line 7Ø selects a random number between Ø and 2 and stores it to the variable TMP. The . in front of TMP tells BASM to give the location of TMP to the RND program, so that it will know where to store the results. Using RND to generate random numbers is slower than using RAND in line 6Ø, but RND gives you more control over the range of numbers.

Line 8Ø adds the random number Ø,1,or2 to the X location, then subtracts 1. This is the equivalent of adding -1,Ø,or1, producing the symmetrical random motion.

Line 9Ø checks for collision with the right side of the screen.

Line 1ØØ checks for collision with the left side of the screen. Notice that 255 can be thought of as -1.

Lines 11Ø to 14Ø do the same as lines 7Ø to 1ØØ for the Y direction.

Line 15Ø plots a dot of the color selected in line 6Ø at the new location.

Line 16Ø is the bottom of the WHILE loop started in line 5Ø.

Line 17Ø closes the graphics channel opened in line 3Ø. Graphics commands always use channel 7, and channel 7 should be closed when you are done with graphics. If you use any other channel, you should use FILE 7 before doing any graphics INPUT/OUTPUT.

Line 18Ø switches back to the normal I/O channel (E:) before returning to BASM.

Line 19Ø returns to BASM.

Line 2ØØ creates the variables used in this

program.

THE BASIC STATEMENTS

Before we get into the actual BASIC statements, let's define a few terms as they relate to the BASM BASIC Compiler.

Variables: In BASM, variable names can be from one to 30 characters. The first character must be a capital letter. The rest of the characters in the variable name can be any digit or capital letter or the underline (_). The underline can be used as a connector between two or more words to form one variable name.

Forbidden variable names: There are some words and symbols that should not be used as variable names. For a complete list of these, see the chapter on "Reserved Names."

Locations: With similar restrictions to variables, locations have the same function in BASM that line numbers have in ATARI BASIC. A location is created by typing its name at the beginning of the line (in column 1) and must have at least one space between it and the statement that follows. A location definition may occur alone on a line. A line without a location definition must be preceded by at least one space.

Constants: BASM uses several types of arithmetic constants:

Decimal: A decimal constant is an unsigned integer number that has a value of from 0 to 255. Examples: 173, 34, 235.

Hexadecimal: A hexadecimal constant must start with the $ symbol followed by

digits (Ø through 9) and/or letters (A through F) that evaluate from $Ø to $FF. Examples: $FØ, $F, $4C.

Octal: An octal constant must start with the @ symbol followed by digits (Ø through 7) that evaluate from @Ø to @377. Examples: @351, @5, @127.

Binary: A binary constant must start with the % symbol, followed by 1 to 8 digits consisting of Ø's and 1's, that evaluate from %Ø to %11111111. Examples: %11Ø, %11ØØ1, %1111ØØ1Ø.

Characters: A character constant is a single character enclosed by single quotes (').

Operators: Used in an expression to modify the current value.

+	Unsigned binary addition
-	Unsigned binary subtraction
AND	Bit-wise logical AND
OR	Bit-wise logical OR
XOR	Bit-wise logical XOR (EOR)
->	Assigns the current value to the following variable

Expressions: An expression is a sequence of byte variables, byte constants, and operators strung together. Expressions are evaluated from the left to the right. No parentheses are used. Also, each part of an expression must be separated by at least one space. Expressions always evaluate to a value between Ø and 255.

Conditions: The following symbols are used as conditions in BASM:

= (equal)

<> (not equal)
> (greater than)
< (less than)
>= (greater than or equal to)
<= (less than or equal to)

Nonexecutable Statements:
A nonexecutable statement must not be in the program flow, but must exist somewhere in the program. This is contrary to the related Atari BASIC statements which must be executed to function. This is something inherent in the concept of compilation and assembly language, and it is not peculiar to BASM. It is the programmer's responsibility to make sure that nonexecutable statements are not executed.

Incorrect sequence in BASM:

```
PRINT "HELLO"
LET Q = Z + 1
DIM Q , Z
```

In all BASIC statements: Whenever a space is specified, several spaces may be inserted. Whenever commas are specified, the commas themselves must be separated by space(s), in order to eliminate possible confusion with certain 65Ø2 addressing modes that use commas.

BASM I/O statements are similar to Atari BASIC Input/Output statements. The largest difference is that the file number is selected by the FILE statement, the OPEN statement, and the CLOSE statement. From then on, that I/O channel is used for Input/Output until otherwise selected. Also, the predefined byte variable STATUS contains the status from that operation.

The statements from GR, the BASM Graphics Library, are similar to Atari BASIC

graphics statements, except that the I/O channel 7 must be maintained as selected during graphics Input/Output operations. Selecting other channels is not harmful to graphics statements, but the channel must be selected back to channel 7 before graphics Input/Output operations may be executed.

On the following pages are the BASIC statements. With each statement is documentation, telling whether or not it is executable, which library file (if any) contains it, the format of the statement, a description of the statement, a few examples of the statement, and a short but functional program using that statement.

BINPUT (Executable, IO library statement)

Format: BINPUT .variable

Inputs one line (terminated by EOL) from the currently selected I/O channel. Converts this from ATASCII decimal format to one byte binary. It then places that value in the specified byte variable. Leading zeros are not needed. Gives a value of Ø for an empty line.

```
BINPUT .COUNT
BINPUT .LINE_NO
```

```
1Ø PRINT "HOW MANY 'HELLO'S' DO YOU WANT?";
2Ø BINPUT .MAX
3Ø FOR NDX = 1 TO MAX
4Ø  PRINT "HELLO"
5Ø NEXT NDX
6Ø RETURN
7Ø DIM NDX , MAX
```

BPRINT (Executable, IO library statement)

Format: BPRINT expression

Converts the byte value from the expression to ATASCII decimal and outputs it to the currently selected I/O channel. There are no leading Ø's or spaces, and it does not append an EOL.

```
BPRINT 5
BPRINT CHAR + 'Ø' AND $7F
```

```
1Ø FOR NDX = 1 TO 1ØØ
2Ø  BPRINT NDX
3Ø  PRINT ""
4Ø NEXT NDX
5Ø RETURN
6Ø DIM NDX
```

CLOSE (Executable, IO library statement)

Format: CLOSE expression

First selects the I/O channel given in the expression (Ø-7), then closes that I/O channel.

```
CLOSE Ø
CLOSE CURRENT_CHANNEL
```

```
1Ø PRINT "SOURCE FILE?";
2Ø INPUT SRC$
3Ø TRAP .NOTHING
4Ø OPEN 1 , 4 , Ø , SRC$
5Ø WHILE
6Ø  FILE 1
7Ø  GET .CHR
8Ø  IF STATUS > 127 GOTO DMPDONE
9Ø  FILE Ø
1ØØ  PUT CHR
11Ø ENDWHILE
12Ø;
13ØNOTHING RETURN
14Ø;
15ØDMPDONE
16Ø CLOSE 1
17Ø FILE Ø : RETURN
18Ø DIM CHR , SRC$(1ØØ)
```

COLOR (Executable, GR library statement)

Format: COLOR expression

This statement selects the color to be used in graphics statements.

```
COLOR Ø
COLOR LAST_COL
```

```
1ØRAND=$D2ØA
2ØCONSOL=$DØ1F
3Ø;
4Ø GRAPHICS 7+16
5Ø REM LOOP UNTIL START PRESSED
6Ø WHILE CONSOL AND 1 <> Ø
7Ø  COLOR RAND AND 3
8Ø  PLOT Ø , RAND AND $7F , RAND AND $3F
9Ø ENDWHILE
1ØØ CLOSE 7
11Ø FILE Ø
12Ø RETURN
```

DATA (Nonexecutable, built-in)

Format: DATA constant , constant , constant ...

The DATA statement is a statement that allows you to put constant data into the middle of your program. This is very useful for creating look-up tables without having to load them into arrays as in Atari BASIC. Simply put a location definition in front of the DATA statement, and you may index into it. Any normal constants may be used in the DATA statement, separated by commas. You may also use string constants in a DATA statement, surrounded by double quotes. Note that no string termination character ($00) is inserted. You get only the characters between the quotes.

```
 DATA 1 , 5 , 138 , 'x'
HEXTABL DATA "0123456789ABCDEF"
```

```
10 GOTO SKIPIT
20 DEF PRHEX HEXDATA
30  LDA HEXDATA
40  LSR A : LSR A : LSR A : LSR A
50  TAY
60  PUT HEXTABLE,Y
70  LET HEXDATA AND $F : TAY
80  PUT HEXTABLE,Y
90  RETURN
100HEXTABLE DATA "0123456789ABCDEF"
110  DIM HEXDATA
120 ENDDEF PRHEX
130SKIPIT
140 FOR NDX = 0 TO $80
150  PRHEX NDX
160  PUT #EOL
170 NEXT NDX
180 RETURN
190 DIM NDX
```

DEF (Nonexecutable, built-in)

Format: DEF statement parameter , parameter , parameter ...

The DEF statement creates a new statement. The parameters that are defined for the new statement must have been dimensioned elsewhere in your program. The DEF statement differs from BASIC in that it defines a statement rather than a function. A statement is used on a line by itself, rather than in an expression. The ENDDEF statement is used to end the definition of a statement. The DEF statement is followed by at least one space, followed by the name of the statement you are defining, which is followed again by at least one space, after which comes the first parameter, followed by a comma, second parameter, comma, third parameter, etc. There is no inherent limit to the number of parameters. For normal byte parameters, a BASM expression may be used for the parameter upon execution (not in the definition). A statement must be defined in the program before it can be used. This refers to program sequence from "top" to "bottom" rather than the sequence of execution. When the statement is used (executed), the name of the statement is followed by the parameters in the sequence they appear in the DEF statement. Besides normal byte variables, there are three special data types allowed in the parameter list.

Word parameter - add a % to the end of the parameter name. When the statement is executed, a word (two bytes) will be transferred to the parameter variable.

Address parameter - add a . to the front of the parameter name. This is similiar to the

word parameter in that it transfers two bytes; however, the address parameter will obtain the the address (location) of the name given in the execution of the statement.

String parameter - add a $ to the end of the parameter name. This will transfer an entire string to the parameter variable. A string is defined as a sequence of ATASCII characters up to 254 long, followed by the null ($ØØ) character.

You may not pass constants for word or address parameters, but you may pass a string constant by surrounding a sequence of characters by double quotes ("). BASM will automatically append an end of line and a null to your string constant. The end of line character may be suppressed by adding a semicolon after the closing quote. Note that the semicolon does not replace a comma in multiple parameter statements.

```
DIM VAL
DEF PRHEX VAL
 LET VAL = VAL + 'Ø'
 IF VAL > '9' THEN LET VAL = VAL + 7
 PUT VAL
 RETURN
ENDDEF PRHEX
...
PRHEX 9
PRHEX ZZZ

DIM PARM1% , PARM2 , .PARM3
DEF XLAC PARM1% , PARM2 , .PARM3
 ...
ENDDEF XLAC
...
XLAC 1Ø , VCNT + 'Ø' , .ZCNT
XLAC X% , Y , .Z

DIM LINE$(1ØØ)
DEF PUT_LINE LINE$
```

```
  PRINT "LINE = ";
  PRINT LINE$
  RETURN
ENDDEF GET_LINE
...
PUT_LINE LINE_X$
PUTLINE "GREETINGS"
```

```
10 GOTO SKIPIT
20 DEF PRHEX HEXDATA
30  LDA HEXDATA
40  LSR A : LSR A : LSR A : LSR A
50  TAY
60  PUT HEXTABLE,Y
70  LET HEXDATA AND $F : TAY
80  PUT HEXTABLE,Y
90  RETURN
100HEXTABLE DATA "0123456789ABCDEF"
110  DIM HEXDATA
120 ENDDEF PRHEX
130;
140 DIM WDVAL% , PROMPT$(100)
150 DEF PRWORD PROMPT$ , WDVAL%
160  PRINT PROMPT$
170  PRHEX WDVAL+1
180  PRHEX WDVAL
190  PUT #EOL
200  RETURN
210 ENDDEF PRWORD
220;
230SKIPIT
240CON=$D01F
250 LET 0 -> PNT -> PNT+1
260 WHILE CON AND 1 <> 0
270  PRWORD "THE VALUE OF PNT = "; , PNT%
280  LET PNT = PNT + 1
290  LDA PNT+1 : ADC #0 : STA PNT+1
300 ENDWHILE
310 RETURN
320 DIM PNT%
```

DIM (Nonexecutable, built-in)

Format: DIM variable , variable , variable ...

The DIM statement creates a BASM variable and allocates space for it. The DIM statement is followed by a variable name, comma, second variable name, comma, third variable name, etc. This allocates one byte for each variable as it creates that variable. You may allocate more space per variable by following the variable name with a constant number enclosed in parentheses. This allocates that number of bytes for that variable. Special data type variables may also be "DIM"ed for use with "DEF"ed statements only. The format of the special data type variable is the same as in the DEF statement. However, the word and address data types may not utilize the "(constant)" feature to increase storage.

```
DIM LINE(100) , CNT
DIM LINE1$(100)
DIM PNT% , .PNT1 , VAL

10 PRINT "WORD TO BE REPEATED?";
20 INPUT LINE$
30 PRINT "NUMBER OF TIMES TO REPEAT?";
40 BINPUT .MAX
50 FOR NDX = 1 TO MAX
60  PRINT LINE$
70 NEXT NDX
80 RETURN
90 DIM NDX , MAX LINE$(100)
```

DRAWTO (Executable, GR library statement)

Format: DRAWTO word variable , expression

This draws a line from the last POSITIONED or PLOTTED point on the screen to the X,Y location indicated. The first parameter is the X location and may be substituted with a Ø , expression if a byte value is desired. The second parameter is the Y location.

```
DRAWTO XVAL% , YVAL
DRAWTO Ø , X , Y
DRAWTO Ø , 1Ø , 4Ø
```

```
1ØCON=$DØ1F
2ØRAND=$D2ØA
3Ø GRAPHICS 7+16
4Ø PLOT Ø , Ø , Ø
5Ø WHILE CON AND 1 <> Ø
6Ø  COLOR RAND AND 3
7Ø  DRAWTO Ø , RAND AND $7F , RAND AND $3F
8Ø ENDWHILE
9Ø CLOSE 7
1ØØ FILE Ø
11Ø RETURN
```

FILE (Executable, IO library statement)

Format: FILE expression

Selects current I/O channel from the value given in the expression. The expression should evaluate from between Ø and 7.

```
FILE Ø
FILE CHANNEL_NO - 1
```

```
1Ø PRINT "FILE NAME?";
2Ø INPUT SRC$
3Ø OPEN 1 , 4 , Ø , SRC$
4Ø OPEN 2 , 8 , Ø , "P:"
5Ø TRAP .ZIP
6Ø WHILE
7Ø  FILE 1
8Ø  GET .CHR
9Ø  IF STATUS > 127 THEN
1ØØ   CLOSE 1 : CLOSE 2
11Ø   FILE Ø
12Ø   RETURN
13Ø  ENDIF
14Ø  FILE 2
15Ø  PUT CHR
16Ø ENDWHILE
17Ø DIM CHR , SRC$(1ØØ)
18Ø DEF ZIP
19Ø  RETURN
2ØØ ENDDEF ZIP
```

FILL (Executable, GR library statement)

Format: FILL word variable , expression , word variable , expression

This statement is similar to the Atari BASIC XIO FILL statement. It draws a line between two points and fills everything to the right of that line with the currently specified color until it meets a nonbackground color. The first parameter is the first X position and may be substituted with a Ø , expression if a byte value is desired. The second parameter is the Y value of the first point. The third parameter is the X position of the second point, which also may be substituted with a Ø , expression if a byte value is desired. The fourth parameter is the Y value of the second point.

```
FILL Ø , 1Ø , 1Ø , Ø , 1Ø , 4Ø
FILL X1% , Y1 , X2% , Y2
```

```
1ØCON=$DØ1F
2Ø GRAPHICS 7+16
3Ø PLOT Ø , 8Ø , 1Ø
4Ø DRAWTO Ø , 8Ø , 8Ø
5Ø FILL Ø , 1Ø , 1Ø , Ø , 1Ø , 8Ø
6Ø WHILE CON AND 1 <> Ø : ENDWHILE
7Ø CLOSE 7
8Ø FILE Ø
9Ø RETURN
```

FOR (Executable, built-in)

Format: FOR variable = expression to expression

The first expression is evaluated once at the beginning of the loop. That value is put into the variable. Each time it loops it adds one to the variable until it is greater than the second expression. The second expression is evaluated each time in the loop. Note that if the first expression is greater than the second expression the loop is never executed. The loop is terminated by the "NEXT" instruction. The syntax of the NEXT instruction is the word NEXT followed by the variable name mentioned in the FOR statement. The two expressions in the FOR statement must evaluate from between Ø and 254. It is legal to exit from the middle of a FOR/NEXT loop. The FOR/NEXT statements do not alter the system stack.

```
FOR NDX = CNT + 12 TO 254
 FOR NDX1 = 1 TO CNT
 NEXT NDX1
NEXT NDX
```

```
1Ø GOTO SKIPIT
2Ø DEF WAIT WAIT_VAL
3Ø  FOR WAIT_NDX = Ø TO WAIT_VAL
4Ø   FOR WAIT_NDX1 = Ø TO 254
5Ø   NEXT WAIT_NDX1
6Ø  NEXT WAIT_NDX
7Ø  RETURN
8Ø  DIM WAIT_VAL , WAIT_NDX , WAIT_NDX1
9Ø ENDDEF WAIT
1ØØSKIPIT
11Ø FOR NDX = Ø TO 1ØØ
12Ø  WAIT  NDX
13Ø  BPRINT NDX : PUT #EOL
14Ø NEXT NDX
15Ø RETURN
```

```
16Ø DIM NDX
```

GET (Executable, IO library statement)

Format: GET .variable

Inputs one byte from the currently selected I/O channel and leaves it in the specified byte variable.

```
GET .TMP
GET .CHAR
```

```
10 OPEN 1 , 4 , 0 , "K:"
20 OPEN 2 , 8 , 2 , "S:"
30 LET CHR = 0
40 WHILE CHR <> 'Q'
50  FILE 1
60  GET .CHR
70  FILE 2
80  PUT CHR + 128
90 ENDWHILE
100 CLOSE 1 : CLOSE 2
110 FILE 0
120 RETURN
130 DIM CHR
```

GOSUB (Executable, built-in)

Format: GOSUB location
or
GOSUB statement parameter , parameter , parameter ...

The GOSUB in BASM is similar to the GOSUB in Atari BASIC. The word GOSUB is followed by a location reference. If you follow the GOSUB statement with a statement name, you may pass parameters to that statement.

```
GOSUB FIX_LOG
GOSUB PRINT "HELLO"
```

```
10 FOR NDX = 0 TO 100
20  GOSUB SHOWIT
30 NEXT NDX
40 RETURN
50SHOWIT
60 PRINT "NDX = ";
70 BPRINT NDX
80 PUT #EOL
90 RETURN
100 DIM NDX
```

GOTO (Executable, built-in)

Format: GOTO location

The word GOTO is followed by a location reference, similar to Atari BASIC.

```
 GOTO ERR12

1ØLOOP PUT 'X'
2Ø LDA $DØ1F : IF AND 1 = Ø THEN RETURN
3Ø GOTO LOOP
```

GRAPHICS (Executable, GR library statement)

Format: GRAPHICS expression

Sets graphics mode selected by expression. The value of the expression acts the same as in the Atari BASIC statement GRAPHICS. Selects channel 7 for I/O to the graphics screen.

```
GRAPHICS MODE_NUMBER + 4
GRAPHICS 8+16
```

```
10CON=$D01F
20 GRAPHICS 8+16
30 COLOR 1
40 SETCOLOR 2 , 0 , 0
50 SETCOLOR 1 , 0 , 15
60 FOR NDX = 0 TO 85
70  PLOT 0 , 0 , 0
80  DRAWTO 0 , 159 , NDX + NDX
90 NEXT NDX
100 WHILE CON AND 1 <> 0 : ENDWHILE
110 CLOSE 7
120 FILE 0
130 RETURN
140 DIM NDX
```

IF (Executable, built-in)

There are three formats to the IF statement.

Format #1: IF expression condition variable/constant THEN statement.

This will execute the statement or statements until the end of the line. These statements themselves may also be IF statements.

```
IF B >= X THEN IF C = Ø THEN STOP
IF Q + 1 = Z THEN RETURN
```

```
1Ø PRINT "NUMBER?";
2Ø BINPUT .XX
3Ø IF XX > 1ØØ THEN PRINT "> 1ØØ"
4Ø IF XX = 1ØØ THEN PRINT "= 1ØØ"
5Ø IF XX < 1ØØ THEN PRINT "< 1ØØ"
6Ø RETURN
7Ø DIM XX
```

Format #2: IF expression condition variable/constant GOTO location.

This is more efficient than using IF...THEN GOTO location, because it generates a direct branch in Assembly language. However, the location reference must be within + or - about 12Ø bytes of the IF statement.

```
IF CNT = Ø GOTO LOC
```

```
1ØCON=$DØ1F
2Ø PRINT "TEXT?";
3Ø INPUT TEXT$
4Ø LDY #Ø
```

```
5Ø IF TEXT = 'A' GOTO XA
5Ø IF TEXT = 'B' GOTO XB
6Ø IF TEXT = 'C' GOTO XC
7Ø PRINT "HUH?"
9Ø RETURN
1ØØXA PRINT "PROGA" : RETURN
11ØXB PRINT "PROGB" : RETURN
12ØXC PRINT "PROGC" : RETURN
13Ø DIM TEXT$(1ØØ)
```

Format #3: IF expression condition variable/constant THEN (THEN is optional)

This form allows you to put many statements and lines in the IF statement. This statement is terminated by the ENDIF statement. Everything between the IF statement and the ENDIF statement will be executed if the condition is true. You may also place an ELSE between the IF statement and the ENDIF statement. In this case, if the condition is true, then the portion between the IF and the ELSE is executed. If the condition is not true, then the portion between the ELSE and the ENDIF is executed.

```
 IF XPNT <> Ø THEN
  LET XPNT = CNT + 5
  RETURN
 ENDIF

 IF XPNT = CNT
  RETURN
 ELSE
  LET CNT = 12
 ENDIF
```

```
1ØCON=$DØ1F
2ØRAND=$D2ØA
3Ø WHILE CON AND 1 <> Ø
4Ø  IF RAND AND 1 <> Ø THEN
5Ø   PRINT "YES"
```

```
60  ELSE
70    PRINT "NO"
80  ENDIF
90 ENDWHILE
100 RETURN
```

INPUT (Executable, IO library statement)

Format: INPUT .string

Inputs one line (terminated by EOL) from the currently selected I/O channel, and stores it in the specified string variable. Appends a BASM (NULL) termination character after the EOL character.

```
INPUT .LINE
INPUT LINE1$
```

```
10 INPUT LINE$
20 LDY #0
30 WHILE LINE,Y <> #EOL : INY : ENDWHILE
40 LET LINE,Y = ' '
50 FOR NDX = 1 TO 100
60  PRINT LINE$
70 NEXT NDX
80 RETURN
90 DIM LINE$(100) , NDX
```

LET (Executable, built-in)

Format: LET variable = expression

This format will evaluate the expression and put the results into the variable. Unlike Atari BASIC, the word LET is not optional. An alternate form of the LET instruction is LET followed by an expression. In this case it will evaluate the expression without assigning it to a variable.

```
LET CNT = CNT + 1
LET LOC+1 AND $FØ -> DST
```

```
1ØAUDF1=$D2ØØ
2ØAUDC1=$D2Ø1
3Ø LET AUDC1 = $AF
4Ø FOR COUNT = 1 TO 1ØØ
5Ø  FOR PITCH = Ø TO 254
6Ø   LET AUDF1 = PITCH
7Ø  NEXT PITCH
8Ø NEXT COUNT
9Ø LET AUDC1 = Ø
1ØØ RETURN
11Ø DIM COUNT , PITCH
```

LOCATE (Executable, GR library statement)

Format: LOCATE word variable , expression , .byte variable

This statement obtains the color number of the specified point on the graphics screen. The first parameter is the X position and may be substituted by a Ø , expression if a byte value is desired. The second parameter is the Y position, and the third parameter is the location of the byte variable you choose to put the color in.

```
LOCATE Ø , 1Ø , 2Ø , .VAL
LOCATE X% , Y , .POINT_COLOR
```

```
1ØCON=$DØ1F
2ØRAND=$D2ØA
3Ø GRAPHICS 6+16
4Ø LET Ø -> XL -> YL
5Ø WHILE CON AND 1 <> Ø
6Ø  LET RAND AND 3 : TAY
7Ø  LET XL + DIR,Y -> XL
8Ø  LET RAND AND 3 : TAY
9Ø  LET YL + DIR,Y -> YL
1ØØ  IF XL = $FF THEN LET XL = Ø
11Ø  IF YL = $FF THEN LET YL = Ø
12Ø  IF XL = 16Ø THEN LET XL = 159
13Ø  IF YL = 96 THEN LET YL = 95
14Ø  LOCATE Ø , XL , YL , .COL
15Ø  COLOR COL XOR 1
16Ø  PLOT Ø , XL , YL
17Ø ENDWHILE
18Ø CLOSE 7 : FILE Ø
19Ø RETURN
2ØØ DIM XL , YL , COL
21ØDIR .BYTE 1,$FF,Ø,Ø
```

OPEN (Executable, IO library statement)

Format: OPEN expression , expression , expression , .string variable / constant

Selects and opens the specified I/O channel. Similar to Atari BASIC OPEN statement. The first expression is the I/O channel (Ø-7). The second expression specifies the type of activity that I/O channel will be doing (input, output, update, etc.). The third expression is the I/O channel's auxiliary byte, and in most cases will be Ø. The fourth parameter (the string) specifies the I/O device and perhaps file name. Should be an ATASCII string terminated by EOL.

```
OPEN 7 , 8 , Ø , "TEST.SRC"
OPEN 3 , 4 , Ø , LINE$
OPEN 4 , 12 , Ø , "E:"
```

```
1Ø PRINT "NEW FILE NAME?";
2Ø INPUT LINE$
3Ø TRAP .DONE
4Ø PRINT "PRESS BREAK WHEN DONE"
5Ø OPEN 1 , 8 , Ø , LINE$
6Ø WHILE
7Ø  FILE Ø
8Ø  INPUT LINE$
9Ø  FILE 1
1ØØ  PRINT LINE$
11Ø ENDWHILE
12ØDONE
13Ø CLOSE 1 : FILE Ø
14Ø STOP
15Ø DIM LINE$(1ØØ)
```

PLOT (Executable, GR library statement)

Format: PLOT word variable , expression

This statement plots one point on the graphics screen. The first parameter is the X position and may be substituted by a Ø , expression if a byte value is desired. The second parameter is the Y position.

```
PLOT X% , Y
PLOT Ø , 1Ø , 2Ø
```

```
1ØCON=$DØ1F
2Ø GRAPHICS 7+16
3Ø FOR NDX = Ø TO 95
4Ø  PLOT Ø , NDX + 63 , NDX
5Ø  PLOT Ø , NDX + 63 , 95 - NDX
6Ø NEXT NDX
7Ø WHILE CON AND 1 <> Ø : ENDWHILE
8Ø CLOSE 7 : FILE Ø
9Ø RETURN
1ØØ DIM NDX
```

POSITION (Executable, GR library statement)

Format: POSITION word variable , expression

This positions the perhaps invisible graphics cursor to that position on the graphics screen. The first parameter is the X position and may be substituted by a Ø , expression if a byte value is desired. The second parameter is the Y position.

```
POSITION X% , Y
POSITION Ø , 1Ø + LOC , LOCY
```

```
1ØCON=$DØ1F
2ØRAND=$D2ØA
3Ø GRAPHICS 2+16
4Ø WHILE CON AND 1 <> Ø
5Ø  POSITION Ø , RAND AND $F , RAND AND 7
6Ø  PUT RAND
7Ø ENDWHILE
8Ø CLOSE 7 : FILE Ø
9Ø RETURN
```

PRINT (Executable, IO library statement)

Format: PRINT string variable / constant

Outputs the string to the selected I/O channel. Does not automatically add an end of line. If desired, the end of line character should be included at the end of the string. See the DEF statement for more information on strings.

```
PRINT LINE$
PRINT "Hello"
PRINT "GOOD BYE";
```

```
10CON=$D01F
20 PRINT "TEXT?";
30 INPUT LINE$
40 WHILE CON AND 1 <> 0
50  PRINT LINE$
60 ENDWHILE
70 RETURN
80 DIM LINE$(100)
```

PUT (Executable, IO library statement)

Format: PUT expression

Outputs 1 byte to the currently selected I/O channel.

```
PUT #EOL
PUT CHAR
PUT 'x'
```

```
10 FOR NDX = 0 TO 25
20  PUT NDX + 'A'
30  PUT ' '
40  IF NDX AND $F = $F THEN PUT #EOL
50 NEXT NDX
60 PUT #EOL
70 RETURN
80 DIM NDX
```

REM (Executable, built-in)

Format: REM anything

Causes BASM to skip the entire rest of the line, ignoring the : separator. Does not affect final code size or speed.

```
REM This section is to be removed
REM THE : IS ALLOWED HERE
```

```
10 REM THIS IS A DEMONSTRATION
20;
30CON=$D01F : REM CONTROL HARDWARE
40 WHILE CON AND 1 <> 0 : REM LOOP UNTIL
START PRESSED
50  BPRINT CNT : REM PRINT CNT
60  PUT #EOL : REM EOL = RETURN
70  INC CNT : REM CNT = CNT + 1
80 ENDWHILE
90 RETURN : REM TERMINATE EXECUTION
100 DIM CNT : REM CREATE CNT
```

RETURN (Executable, built-in)

Format: RETURN

The RETURN statement in BASM is used in exactly the same way as Atari BASIC. A return from the main sequence of your program will return to BASM Editor or the Atari operating system.

```
 RETURN

10 FOR NDX = 1 TO 10
20  GOSUB SHOWIT
30 NEXT NDX
40 RETURN
50 DIM NDX
60SHOWIT PRINT "THE VALUE = ";
70 BPRINT NDX
80 PUT #EOL
90 RETURN
```

SETCOLOR (Executable, GR library statement)

Format: SETCOLOR expression , expression , expression

The first parameter selects which color is to be altered. The second parameter selects the new hue, and the third parameter selects the new brightness.

```
SETCOLOR 1 , 4 , 15
SETCOLOR NDX + 1 , COLR , BRI
```

```
10CON=$D01F
20RAND=$D20A
30 WHILE CON AND 1 <> 0
40  SETCOLOR 2 , RAND AND $F , RAND AND $F
50 ENDWHILE
60 SETCOLOR 2 , 8 , 2
70 RETURN
```

SOUND (Executable, GR library statement)

Format: SOUND expression , expression , expression , expression

This is similar to the SOUND statement in Atari BASIC. The first parameter selects which sound is to be altered; the second parameter is the pitch; the third parameter is the distortion; and the fourth parameter is the volume.

```
SOUND 1 , NDX , 10 , 14
SOUND NDX + 1 , 0 , 0 , 0
```

```
10 FOR NDX = 1 TO 100
20  FOR PITCH = 10 TO 200
30   SOUND 0 , PITCH , 10 , 10
40   SOUND 1 , 210 - PITCH , 10 , 10
50  NEXT PITCH
60 NEXT NDX
70 SOUND 0 , 0 , 0 , 0
80 SOUND 1 , 0 , 0 , 0
90 RETURN
100 DIM NDX , PITCH
```

STOP (Executable, BASM.LIB library statement)

Format: STOP

This statement stops the program and returns control to BASM or the Atari operating system, regardless of the condition of the system stack. If BASM.LIB was obtained from BASM.HI (default), then STOP will return program execution to BASM. If BASM.LIB was obtained from MASM.LO, then STOP will return execution to Atari DOS. If you maintain the stack (proper subroutine nesting), then the RETURN statement will return you to either, if executed from the main sequence.
STOP

```
10 PRINT "FILE NAME?";
20 TRAP .DONE
30 INPUT LINE$
40 OPEN 1 , 4 , 0 , LINE$
50 WHILE
60  FILE 1 : INPUT LINE$
70  FILE 0 : PRINT LINE$
80 ENDWHILE
90 DEF DONE
100  CLOSE 1
110  FILE 0
120  STOP
130 ENDDEF DONE
140 DIM LINE$(100)
```

TR (Executable, TRACE library statement)

Format: TR expression

This statement is the TRACE statement. When this statement is executed, the program will switch displays, display the trace number you specified as the first parameter, wait for you to press the START key, and then continue with the execution of the program after restoring the previous screen. Note that this statement will not alter your graphics screen during execution.

```
TR 1
TR 100
TR NDX + 12
```

```
10CON=$D01F
20RAND=$D20A
30 GRAPHICS 7+16
40 WHILE CON AND 2 <> 0 : REM LOOP UNTIL
SELECT
50  LET XL = RAND AND $7F
60  LET YL = RAND AND $3F
70  LOCATE 0 , XL , YL , .TMP
80  IF TMP <> 0 THEN
90   TR XL
100   TR YL
110   TR TMP
120  ENDIF
130  COLOR RAND AND 3
140  PLOT 0 , XL , YL
150 ENDWHILE
160 CLOSE 7 : FILE 0 : RETURN
170 DIM XL , YL , TMP
```

TRAP (Executable, IO library statement)

Format: TRAP .location

Designates the user error handling subroutine that is executed when an I/O error is detected. The location may be a program location or a user statement with no parameters. All I/O operations update the STATUS variable, which is dimensioned in the IO library and is the status byte from the I/O device.

```
TRAP .ERROR
TRAP .DO_NOTHING
```

```
10CON=$D01F
20 TRAP .ERR
30 PRINT "FILE NAME?";
40 INPUT LINE$
50 OPEN 1 , 4 , 0 , LINE$
60 TRAP .DONE
70 WHILE
80  FILE 1
90  INPUT LINE$
100  FILE 0
110  PRINT LINE$
120 ENDWHILE
130 DIM LINE$(100)
140DONE
150 CLOSE 1 : FILE 0 : GOTO WAIT
160ERR
170 CLOSE 1 : FILE 0
180 PRINT "BAD FILE NAME"
190WAIT WHILE CON AND 1 <> 0 : ENDWHILE
200 STOP
```

WHILE (Executable, built-in)

Format: WHILE expression condition variable/constant

WHILE is terminated by the ENDWHILE statement. This creates a WHILE loop. All statements between the WHILE and the ENDWHILE are executed as long as the WHILE condition remains true. If the WHILE condition is false to begin with, then the statements between the WHILE and the ENDWHILE are never executed.

An alternate form of the WHILE statement is WHILE by itself with no condition. This produces an infinite loop. However, it is legal to exit from the middle of a WHILE loop.

```
LDY CNT
WHILE LINE,Y <> Ø
 LET DST,Y = LINE,Y
 INY
ENDWHILE

WHILE
 IF LINE,Y = #EOL THEN RETURN
 INY
ENDWHILE
```

```
1Ø INPUT LINE$
2Ø LDY #Ø
3Ø WHILE LINE,Y <> #EOL
4Ø  INY
5Ø ENDWHILE
6Ø LDX #Ø
7Ø WHILE TXT,X -> LINE,Y <> Ø
8Ø  INX : INY
9Ø ENDWHILE
1ØØ PRINT LINE$
11Ø RETURN
12Ø DIM LINE$(1ØØ)
```

```
130TXT DATA " = TEXT" , EOL , 0
```

ASSEMBLY LANGUAGE

Assembly Language in BASM is the standard 6502 format. The Assembly language format consists of the following:

Optional location definition (also called label definition) followed by at least one space followed by the operation code mnemonics (or op code, for short) which may be followed by an operand. If it is followed by an operand, it must be separated by at least one space. This may be followed by a comment. You may have more than one Assembly language statement on a line. Each statement must be separated by a colon, similar to the BASIC format. In fact, BASIC statements and Assembly language statements may be mixed on the same line, using the colon. If you desire a location definition, then the instruction must be the first on the line, and the location definition must begin in column 1.

The location definition is similar to that of BASIC. It essentially allows you to give a name to that position in memory where this instruction occurs.

The op code tells BASM what instruction is to be put at that specific point in the program.

The operand field tells BASM how to further redefine that instruction in terms of which addressing mode, and which particular location in memory that instruction is referencing.

This may be followed by an optional comment. BASM ignores comments. Note, however, that

the comment should not contain the colon character, because BASM would interpret that as the beginning of the next instruction.

ASSEMBLER MNEMONICS

ADC	Add Memory to Accumulator with Carry
AND	AND Accumulator with Memory
ASL	Shift Left
BCC	Branch if Carry Clear
BCS	Branch if Carry Set
BEQ	Branch if Equal Zero
BIT	Test Memory Against Accumulator
BMI	Branch if Minus
BNE	Branch if not Equal Zero
BPL	Branch if Plus
BRK	Break
BVC	Branch if V Flag Clear
BVS	Branch if V Flag Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode Flag
CLI	Clear Interrupt Disable Flag
CLV	Clear V Flag
CMP	Compare Accumulator and Memory
CPX	Compare Register X and Memory
CPY	Compare Register Y and Memory
DEC	Decrement Memory
DEX	Decrement Register X
DEY	Decrement Register Y
EOR	Exclusive-OR Accumulator with Memory
INC	Increment Memory
INX	Increment Register X
INY	Increment Register Y
JMP	Jump to New Location
JSR	Jump to Subroutine
LDA	Load Accumulator
LDX	Load Register X
LDY	Load Register Y
LSR	Shift Right
NOP	No Operation
ORA	OR Accumulator with Memory
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack

PLP	Pull Processor Status from Stack
ROL	Rotate Left
ROR	Rotate Right
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode Flag
SEI	Set Interrupt Disable Flag
STA	Store Accumulator
STX	Store Register X
STY	Store Register Y
TAX	Transfer Accumulator to Register X
TAY	Transfer Accumulator to Register Y
TSX	Transfer Register SP to Register X
TXA	Transfer Register X to Accumulator
TXS	Transfer Register X to Register SP
TYA	Transfer Register Y to Accumulator

ADDRESSING MODES

The addressing modes for the 65Ø2 are defined in the operand field. Not all instructions have all or any addressing modes. An addressing mode is essentially the way a computer tells where in memory to receive and store information.

Operand expressions are made up of operators and terms. No spaces may be inserted between operators and terms.

TERMS

```
lable   - Returns the 16-bit value of
          the lable
>lable  - Returns the high-order byte
          of the lable
<lable  - Returns the low-order byte
          of the lable
number  - Decimal if starts with Ø-9
          Hexidecimal if starts with $
          Octal if starts with @
          Binary if starts with %
          Character if starts with '
*       - Current 16-bit value of
          Location counter
```

OPERATORS

```
+       - Addition
-       - Subtraction
```

```
LINE+1
'C'+1Ø-OFFS
$1ØØ+EOL
>XXXX
<XXXX
%111ØØ111
```

ADDRESSING MODES

Immediate

The immediate addressing mode is specified by placing # in front of the operand expression. The operand expression must evaluate to a value between Ø and 255. In the immediate addressing mode the data specified in the operand is itself the data acted upon. That is to say an ADC #8 would add 8 to the accumulator. This addressing mode is legal in a BASIC expression. It is legal to give a label reference in the immediate mode as long as the label evaluates to a value between Ø and 255. For example, if the label PNT was assigned to be equal to 12, then AND #PNT would "and" 12 to the accumulator.

```
EOR #$Ø1
ADC #EOL
AND #8
```

Memory direct

Direct is specified by having either a number or a label reference in the operand field. In memory direct addressing, the value of the operand tells where in memory to obtain the data to be acted upon. This addressing mode has two forms. If the operand evaluates to between Ø and 255, then the instruction occupies 2 bytes. If the operand evaluates to between 256 and 65535, then the instruction occupies 3 bytes. This addressing mode may be used in a BASIC expression, and is, in fact, the normal variable reference.

```
SBC CNT
ORA 5
LSR SAND+1
```

Implied addressing

Some instructions have no operand at all. This is called implied addressing.

```
TAX
INY
RTS
```

Accumulator addressing

Some instructions may operate on the 65Ø2 A register. This is specified by using the letter A as the operand.

```
LSR A
ASL A
```

Pre-indexed indirect addressing

Specified by (operand expression,X) The operand expression must evaluate to between Ø and 255. The value of the operand expression is added to the current value in the X register. This points to a two byte value in the lower 256 memory locations, which in turn point to the location in memory where the actual data is located. Note that this is the least used 65Ø2 addressing mode. This is because it is probably the least useful. May be used in a BASIC expression.

```
ADC (NDX,X)
LDA (5,X)
```

Post-indexed indirect addressing

Specified by (operand expression),Y The operand expression must evaluate to between Ø and 255. This is the address of two bytes which (after the content of the Y register is added) form a pointer into memory where

the actual data resides. May be used in a BASIC expression.

```
LDA (PNT),Y
STA (5),Y
AND (LINE_PNT+2),Y
```

Indexed addressing

Specified by operand expression,X or operand expression,Y Similar to memory direct, except that the content of the X or Y register is added to the address before it is used. If the operand expression evaluates to between Ø and 255, then the instruction is 2 bytes long. If the operand evaluates to between 256 and 65535, then the instruction is 3 bytes long. May be used in a BASIC expression.

```
LDA VAL_TAB,Y
ADC 5,X
AND $1ØØØ,Y
```

Indirect addressing

Specified by (operand expression) Is used only by the JMP operation. The operand points into memory where a two byte pointer is contained. This value is copied into the program counter.

```
JMP ($EØØØ)
JMP (NEW_EXEC)
```

Relative addressing

Specified by operand expression Is used only by the branch instructions. The operand expression must evaluate to a number within -126 to +129 bytes of the branch instruction.

```
BCS LOOP
```

```
BNE ERROR
```

ASSEMBLY DIRECTIVES

Directives are instructions to BASM rather than generating any code. Except for the = and the *= directives, all directives begin with . (period).

= Directive

Format: label = operand expression

Creates "label" name and assigns the value of the operand expression to it. Note this in this directive, the label is not optional, and must begin in column 1.

```
SYBUF = $600
EOL=$9B
SYBUFX= SYBUF+12
```

*= Directive

Format: *= operand expression

Causes the current program location to be equal to the operand expression. Any labels used in the operand expression must already have been defined. The *= must not begin in column 1.

```
*=$8000
*= *+100
```

.BYTE Directive

Format: .BYTE operand expression,operand expression,operand expression...

Assigns the values of the operand

expressions in memory, in sequence. It evaluates the operand expressions from left to right and allocates 1 byte to each. You must have at least one operand expression, and no spaces are allowed between operand expressions. Each operand expression must evaluate to bewteen Ø and 255. In addition, a sequence of ATASCII characters may be included in the operand list, surrounded by single quotes (').

```
CNT .BYTE Ø
XTAB .BYTE 'NOT',34,$FF
 .BYTE 'Nega ',39,' ',Ø
```

.WORD Directive

Format: .WORD operand expression,operand expression,operand expression...

Assigns the values of the operand expressions in memory, in sequence. It evaluates the operand expressions from left to right and allocates 1 word (two bytes, low order first) to each. You must have at least one operand expression, and no spaces are allowed between operand expressions. Each operand expression must evaluate to bewteen Ø and 65535.

```
CNT .WORD Ø
XTAB .WORD Ø,1ØØØ,$FFFF
 .WORD XTAB,1ØØ
```

.END Directive

Format: .END

This directive is included for compatibility with 65Ø2 assembly code, and has no effect.

```
.END
```

.LST Directive

Format: .LST special operand

Controls the BASM list file. Only one operand is allowed in LST directive.

.LST OFF	Turns off the program listing
.LST ON	Turns on the program lising
.LST CODE	Causes code generated by BASIC to be listed
.LST NOCODE	Causes code generated by BASIC to not be listed
.LST SOURCE	Causes source code to be listed
.LST NOSOURCE	Causes source code to not be listed

The default setting is ON, NOCODE, SOURCE.

.TITLE Directive

Format: .TITLE 'title text'

Sets the title that appears at the top of each page in the listing. The title text must be enclosed by single quotes. The default title is '' (nothing).

```
.TITLE 'Test Program'
```

.EJECT Directive

Format: .EJECT

Ejects the lising page. Does not send form feed to printer, but advances by blank lines.

```
.EJECT
```

.INCL Directive

Format: .INCL 'filename'

Takes the contents of the disk file named filename and includes it in the program as source code at the position where the .INCL occurs. This program segment may itself contain .INCL statements. This "nesting" of files may continue up to 5 deep. If the file is on D1: then no "device:" is required.

```
.INCL 'IO'
.INCL 'D2:SPECIAL'
```

PROGRAMMING BASM

BASM is an OPEN language, in that the syntax is not defined to restrict you to "correct" programming methods. It allows you a great deal of flexibility to do programming "tricks," and to optimize your code. However, this also gives you more responsibility in making sure of program correctness.

BASM makes no distinction between location references, variable names or library commands. Also, in the special data types, the . % $ are trimmed: A$ is the same as A as A% as .A, as far as names are concerned. This means that you could GOTO a variable or use the first byte of a command as data. This makes it very easy for you to define Atari operating system subroutines as commands, among other things.

During command execution, BASM does not check to make sure that your parameters match. Parameters are passed on the system stack and the last byte (or low order half of a word) in the A register. Word parameters pass two bytes, high order first. Address and string parameters pass the address of the variable as two bytes, high order first. The copying of a string is done in the command. It is often useful to use the assembly "=" directive, in order to create the parameter variables in page Ø, rather than using "DIM".

BASM expressions always leave the final result in the 65Ø2 A register. Also, it is legal to leave out all or part of the expression and just use the contents of the A register. Conditions (=,<>,>,etc.) do not alter the A register.

```
IF CHAR = 'A' GOTO ALOC
IF = 'B' GOTO BLOC
IF = 'C' GOTO CLOC
LET - '0' -> ERR_NUM
```

The -> assign can be very useful in IF and WHILE statements. The following will move an entire string.

```
LDY #0
WHILE LINE1,Y -> LINE2,Y <> 0
 INY
ENDWHILE
```

The logical operators are useful for "trimming" your data.

```
AND %00001111 will force bits 4-7 to 0
OR  %00001111 will force bits 0-3 to 1
XOR %00001111 will invert bits 0-3
```

It is important to learn the 6502 addressing modes (see chapter on assembly language). These are the 6502 instructions that are most important to learn in BASM:

```
LDY xx Loads the Y register
LDX xx Loads the X register
STY xx Stores the Y register to memory
STX xx Stores the X register to memory
INY    Adds 1 to the Y register
DEY    Subtracts 1 from the Y register
INX    Adds 1 to the X register
DEX    Subtracts 1 from the X register
INC xx Adds 1 to byte variable
DEC xx Subtracts 1 from byte variable
```

To add two word variables:

```
LET DST = SRC1 + SRC2
LDA SRC1+1 : ADC SRC2+1
```

```
 STA DST+1
```

To subtract two word variables:

```
 LET DST = SRC1 - SRC2
 LDA SRC1+1 : SBC SRC2+1
 STA DST+1
```

To move a word variable:

```
 LET DST = SRC
 LET DST+1 = SRC+1
```

To index into a string variable:

```
 LDY NDX
 LET CHAR = LINE,Y
```

You may use a command before it is defined with the GOSUB statement.

```
 GOSUB SCRL 4 , 5
 DEF SCRL PRM1 , PRM2
```

BASM LIBRARY

BASM has a very simple library system. During compilation, the file BASM.LIB must exist on disk D1:. This file is automatically included (.INCL) at the beginning of your program. It is an ordinary text file and contains some set-up functions, defines some system variables, and contains several .INCL's. You may easily load it into the BASM text work space, modify it, and store it back to BASM.LIB . BASM occupies memory from locations $1EØØ to $5EØØ, so there are two versions of BASM.LIB to locate your program in memory. BASM.HI locates your program at $5EØØ and causes STOP to return to BASM. BASM.LO locates your program at $1EØØ and causes STOP to return your program to the Atari DOS. There is also ASSEM.LIB which can be used if you wish to use BASM as just an assembler. Just copy the files to BASM.LIB to use them. BASM is shipped with BASM.HI copied to BASM.LIB. If you modify BASM.LIB, make sure that .INCL 'TRACE' occurs before any other .INCL's. Also, .INCL 'GR' uses some commands in 'IO'.

There are more library files, not .INCL'ed in the shipped BASM.LIB, but these may be used by adding .INCL statements to BASM.LIB.

FLOAT

The floating point library interfaces to the Atari operating system floating point programs. Floating point variables occupy 6 bytes and are referenced by using the . address modifier. FLOAT uses commands defined in 'IO'.

FADD .SRC1 , .SRC2 , .DST
Adds SRC1 to SRC2 and puts the results in DST

FSUB .SRC1 , .SRC2 , .DST
Subtracts SRC2 from SRC1 and puts the results in DST

FMUL .SRC1 , .SRC2 , .DST
Multiplies SRC1 by SRC2 and puts the results in DST

FDIV .SRC1 , .SRC2 , .DST
Divides SRC1 by SRC2 and puts the results in DST

FSTR .SRC , DST$
Converts the contents of SRC to a BASM string and stores the results at DST

FVAL SRC$, .DST
Converts the contents of the BASM string at SRC to floating point, and stores the results at DST. (Note that SRC may be a string constant)

FINPUT .DST
Inputs one line from the currently selected I/O channel, converts it to floating point format, and stores the results at DST

FPRINT .SRC
Converts the contents of SRC to BASM

string format and outputs the results to the currently selected I/O channel. Does not append an end of line character

FTOW .SRC , .DST
Converts the contents of SRC from floating point to word integer, and stores the results at DST

WTOF .SRC , .DST
Converts the word at SRC to floating point format and stores the results at DST

FEXP .SRC , .DST
Takes the exponent (base 10) of SRC and stores the results at DST

FLOG .SRC , .DST
Takes the logarithm (base 10) of SRC and stores the results at DST

MISC

MISC adds some high speed graphics routines and a random number generator. MISC uses commands defined in IO and GR.

RND SRC , .DST

Finds a random number (byte) between Ø and SRC (inclusive) and store the results at DST. A value of Ø for SRC will cause an infinite loop

GR7

Creates a graphics 7 screen with no split text and clears the screen. Also, sets up variables for PLOT7

PLOT7 XLOC , YLOC

Takes the color selected by the COLOR command and plots it on the graphics screen. Much faster than going through the operating system. XLOC and YLOC are byte variables or expressions

EXTRA

Whenever we write new library files or utility programs, we immediately put them in the distribution disk. Sometimes it takes a while to get the documentation into the manual, so we are including documentation files explaining, the added libraries & utilities. These documentation files will have the extension ".DOC" and may be loaded into the BASM editor workspace as BASM source programs, or printed out as text files.

UTILITIES

Three programs are included with BASM to facilitate program development.

FAST - Speeds up the loading of your program after program development is finished. It does this by loading your program into memory, then saving it back to disk as one block. If you place all of your variable DIMs at the end of your program, then these will not be included in the block, and will not waste disk space. Will normally speed up the program loading time by about a factor of two. This program starts at $5EØØ, but the source code is included, so that you may reposition it in memory.

FILES - Tells you how many disk files may be opened at once, then lets you change the number of disk files which may be opened at once. Returns to Atari DOS. Should be run from DOS. To make the change permanent, use the Atari write system files function.

MONITOR - A simple machine monitor with several commands. Starts at $5EØØ. Numbers are entered as hexadecimal with no leading zeros needed. Numbers may be separated by spaces or commas.

L number , number

Lists memory in hexadecimal from the first number to the second number, inclusive. If the second number is absent, then it lists only one location.

S number , number , number ...

Substitutes memory locations, using the

first number as the starting address, the second number as the first data, the third number as the second data, etc.

, number , number ...

Continues on where the S command left off. Care should be taken to use this only immediately after the S command.

G number

Executes at the given address in memory.

Q

Quit, return to BASM or DOS.

RESERVED NAMES

Reserved names may not be used as variables or location definitions. Note that some of the reserved names are actually statements or variables defined in the system library files BASM.LIB, IO, GR, or TRACE. You should also aviod using any name starting with SY .

A, AND, BINPUT, BPRINT, CLOSE, COLOR, DATA, DEF, DIM, DRAWTO, ELSE, ENDDEF, ENDIF, ENDWHILE, EOL, FILE, FILL, FOR, GET, GOSUB, GOTO, GRAPHICS, IF, INPUT, LET, LOCATE, OPEN, OR, PLOT, POSITION, PRINT, PUT, RETURN, SETCOLOR, SOUND, STATUS, STOP, THEN, TO, TR, TRAP, WHILE, XOR

BASM ERROR MESSAGES

The following are the explanations for the error messages encountered in the BASM compiler:

1 Nesting error of compiler-generated symbols. May be caused in consequence of other syntax errors.

2 Lack of ending quote in string constant.

5 Illegal condition in IF or WHILE.

6 Improper syntax of IF statement.

7 IF-THEN-ENDIF in a single line IF statement.

8 Too many ENDIF statements.

9 Syntax error in the FOR statement.

10 Too many NEXT statements.

11 Syntax error in the DIM statement.

12 Syntax error in the DEF statement.

13 Syntax error in the WHILE statement.

14 Too many ENDWHILE statements.

15 Illegal syntax of the operator in an Assembly language statement.

16 Illegal character in the operand of an Assembly statement.

17 Bad address range in an Assembly statement.

18 Reference to nonexistent label or variable. May be caused in consequence of other syntax errors.

19 Double label or variable definition or second-pass mismatch. Second-pass mismatch may be caused by reference to a page Ø variable before it is defined. It may also be caused by the reference to a defined statement before it is defined.

20 Too many IF statements without matching ENDIF statements.

21 Too many FOR statements without matching NEXT statements.

22 Too many WHILE statements without matching ENDWHILE statements.

23 Too many DEF statements without matching ENDDEF statements.

24 .INCL syntax error.

25 Lack of end quote in the operand of a DATA statement.

All disk errors terminate the compilation. See the disk operator's manual for an explanation of the disk error numbers. However, note that error 161 may occur if too deep of a nesting of the .INCL command is attempted.

NOTICE

Upon receipt of this computer program and associated manual, COMPUTER ALLIANCE grants you a nonexclusive license to execute the enclosed compiler/assembler/editor program named BASM and to make back-up copies of it for your personal use only; and only on the conditions that any media containing a copy or copies of the BASM program are conspicuously marked with the same copyright notice that appear on the original. This BASM program and associated manual are copyrighted. You are prohibited from reproducing, translating, or distributing the BASM program or manual in any unauthorized manner.

However, you ARE AUTHORIZED to sell, reproduce, and transmit to other parties the binary program or programs you may generate using the BASM program. COMPUTER ALLIANCE does not assume any rights to your programs, though they may be developed using the BASM program.

Atari is a registered trademark of Atari, Inc. The use of trademarks or other designations is for reference purposes only.

LIMITED WARRANTY

This software product and the attached instructional materials are sold "AS IS" without warranty as to their performance. The entire risk as to the quality and performance of the computer software program is assumed by the user. The user, and not the manufacturer, distributor or retailer assumes the entire cost of all necessary service or repair to the computer software program. However, to the original purchaser only, COMPUTER ALLIANCE warrants that the medium on which the program is recorded will be free from defects in materials and faulty workmanship under normal use and service for a period of ninety days from the date of purchase. If during this period a defect in the medium should occur, the medium may be returned to COMPUTER ALLIANCE or to an authorized COMPUTER ALLIANCE dealer, and COMPUTER ALLIANCE will replace or repair the medium at COMPUTER ALLIANCE'S option without charge to you. Your sole and exclusive remedy in the event of a defect is expressly limited to replacement or repair of the medium as provided above. To provide proof that you are the original purchaser, please complete and mail the enclosed Owner Warranty Card to COMPUTER ALLIANCE. If failure of the medium, in the judgement of COMPUTER ALLIANCE, resulted from accident, abuse or misapplication of the medium, then COMPUTER ALLIANCE shall have no responsibility to replace or repair the medium under the terms of this warranty. The above warranties for goods are in lieu of all other expressed warranties and no implied warranties and fitness for a particular purpose or any other warranty obligation on the part of COMPUTER ALLIANCE shall last longer than ninety days. Some states do not allow limitations on how long an implied warranty lasts, so the above limitations may not apply to you. In no event shall COMPUTER ALLIANCE, or anyone else who has been involved in the creation and production of this computer program, be liable for indirect, special, or consequential damages, such as, but not limited to, loss of anticipated profits or benefits resulting from the use of this program, or arising out of any breach of this warranty. Some states do not allow exclusion or limitation of incidental or consequential damages, so the above limitations may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state. COMPUTER ALLIANCE reserves the right to make improvements to this manual and the product described herein at any time and without notice.