

# MISOSYS

## Enhanced BASIC Compiler Development System

Copyright 1986 Philip A. Oliver  
All rights reserved

Reproduction of this manual in any manner, electronic, mechanical, magnetic, optical, chemical, or otherwise, without written permission, is prohibited.

The MISOSYS Enhanced BASIC Compiler product is published by:

MISOSYS, Inc.  
P. O. Box 239  
Sterling, Virginia 22170-0239

[703-450-4181]

**LDOS** is a trademark of Logical Systems, Inc.  
**MICROSOFT** is a trademark of the Microsoft Corp.  
**TRSDOS** is a trademark of Tandy Corp.

## 1 Introduction

### 1.1 Important Note

Certain documentation pertaining to this package may be available after the user manual has gone to press. Consult the file entitled README/TXT for details on additional support material and errata. If you are new to LDOS, read the booklet entitled "Running under LDOS".

### 1.2 Distribution Disks

The TRSDOS 6.x EnhComp Development System is distributed on a 40-track double density data diskette.

The Model I/III EnhComp Development system works on both the Model I and Model III under LDOS 5.x, and TRSDOS 1.3. It is released on a 40 track double density Model III smallDOS system diskette. TRSDOS 1.3 users must use the procedure outlined under TRANSFERRING ENHCOMP TO TRSDOS 1.3 and a two-drive system to transfer the files from the master disk to a working system disk. The master disk is readable by LDOS and DOSPLUS. Model I or III use under a DOS other than LDOS may require patches to one or more of the supplied programs.

### 1.3 General Information

To begin with, the EnhComp BASIC Compiler Development System comprises five files that are on the disk included with this package. These are BC/CMD, CED/CMD, REF/CMD, S/CMD, and SUPPORT/DAT.

BC/CMD is the actual BASIC compiler. It normally produces a directly executable Z80 machine language /CMD file on compilation finish, from a user-supplied source program. This compiled code uses an efficient internal pseudo-code for the most part.

CED/CMD is a special line-oriented editor included should you desire to use it. You can use an editor that you're familiar with if you so choose; however, EnhComp expects its input to be in either pure ASCII form, with line numbers required for every line, or in its own specially tokenized format, which is provided by CED/CMD. In addition to more efficiently storing your source code in memory and on disk because of EnhComp keyword tokenization, CED allows you to merely type 'RUN' to semi-interactively compile and execute (if 0 errors are detected) your current program, returning control to CED on program completion or compiler error abort.

S/CMD is a "supervisor" program required for the interactive 'RUN'. It is a small program that automatically loads and executes CED/CMD when it is itself executed. Although CED can be used without S/CMD invocation, interactive RUNS will be disallowed.

REF/CMD is the utility for generating the reference report.

SUPPORT/DAT is a relocatable library module, in a special format, which contains support subroutines needed for various BASIC instructions and utilities. They are appended as needed to the compiled program, thus assuring that no wasted utilities are included.

These files comprise the fundamental EnhComp compiler system. SUPPORT/DAT must be available on one of your disks during every compile. Compilation will automatically be aborted if SUPPORT/DAT is not available. It is recommended that SUPPORT/DAT reside on a different drive (say, drive 1) than the compiled program destination drive (say, drive 0). This greatly reduces excessive disk drive repositioning during the compilation process.

For the same reason, it is a good idea to separate the source and object files on different disks. If using an interactive editor RUN, you can pre-create TEMP/BAS, which holds your source during compilation, TEMP/CMD, which holds the compiled program, and TEMP/DAT, which holds the optional reference data file, on different drives, to assure this.

EnhComp acts as a translator between high level language, which most people find easiest to program in, to faster Z80 machine language (and pseudo-code), which most people find hard to program with. Sometimes this translation is simple; sometimes it's complex. An experienced assembly language programmer can usually produce more efficient code than a compiler, including the so-called optimizing compilers. Because a "core" of subroutines is included as needed, the size of relatively short EnhComp programs will be around 8-9k larger than the source file. Lacking the time and/or money required to write an assembly program from scratch to duplicate a high level program, a compiler is a good compromise, and is quicker in any case.

#### **1.4 Note on Merchantability**

Neither the author nor the Publisher of EnhComp makes any guarantee as to the fitness of EnhComp, or programs generated by EnhComp, for any particular use, nor do they assume any liability whatsoever for any damages that may arise directly or indirectly through the use of EnhComp and associated material such as this manual, including through programming errors that may be found. Publisher's sole liability shall consist of replacing magnetic media found defective by the buyer upon first testing the distribution diskette. By using EnhComp, you imply acceptance of these terms.

However, the author has gone to the greatest feasible measures for testing the reliability of EnhComp and has confidence that it will work as described herein. Due to the nature of programming, certain errors will probably occur periodically, especially in a program with the complexity of EnhComp. The Publisher would appreciate receiving comments from EnhComp users about bugs found and will make every effort to correct them in future versions, which will be made available to registered purchasers of EnhComp for a nominal fee as they become available.

## 2 Program Compilation

### 2.1 Compilation from CED Editor

The easiest way to compile a source program is to use CED to create an EnhComp program and then type RUN. For a "standard", plain vanilla compilation, it's as easy as an interpretive BASIC RUN, although much slower.

If you have no test program handy, here's one to use. Type 'S' at DOS READY. CED will automatically be loaded. Then, using the same procedure as the TRS80 BASIC editor (i.e., typing all lines verbatim), enter the following.

```
'  
' Draws design on the screen  
'  
CLS  
FOR Y=0 TO 47 STEP 3  
'  
' Plot lines moving in opposite directions from opposing  
' corners  
PLOT S,0,0 TO 127,Y:PLOT S,127,47 TO 0,47-Y  
NEXT  
A$=WINKEY$:END
```

Once you've entered this simple program, simply type RUN and wait for compilation to finish; this should take around a minute and probably less if you're using hard disks or RAM disks.

If TEMP/BAS already exists, the message 'Replacing existing file' will appear; otherwise, 'Creating new file' or something similar will be printed. After your source has been saved to disk (notice that the EnhComp system is usually disk I/O bound), BC/CMD will be loaded.

After the initial message has been printed, the sentence 'PASS #1' will appear. EnhComp is a two-pass compiler, so this is only the first run through your source program. Soon the message 'Appending support subs' will appear, along with the subroutine currently being linked.

Upon completion of the first pass, 'PASS #2' informs you of the start of the last pass. When this is done, and the support routines have been linked in from SUPPORT/DAT, you'll see various information detailing the loading area in memory of the compiled program and the number of bytes required by each data table (this need not concern you at the moment.) If all went well, there will be 0 errors, and TEMP/CMD, which holds the compiled program, will be loaded and executed. After the design has been created, the 'A\$=WINKEY\$' instruction waits for a key to place in A\$; press any key to have CED, and your source code, re-loaded for another round.

Although the programming cycle is somewhat slow, as with almost all floating point, non-trivial compilers, this procedure is much less taxing and irritating than the conventional edit, save, run compiler, ink, execute, etc. cycle.

If things didn't go quite as smoothly as described; that is, if you got some error messages while compiling the program, check your program. If it was the one given, make sure you typed it in correctly. The error codes (summary

given elsewhere in this manual) should help you locate the source of the problem.

If the error was DOS related, an appropriate message will be given, followed by a detailed DOS error message. The supervisor will automatically give an error message if a fatal DOS error occurred (e.g., missing BC/CMD or SUPPORT/DAT).

Note that when using an interactive RUN, and barring a fatal disk error like a missing sector, your current program will be safely in TEMP/BAS should anything go drastically wrong; which can happen in such instances as bad Z80 assembly code in your source file, and so on. Simply re-boot, type 'S', and load in TEMP/BAS using 'L:TEMP/BAS'.

Note that due to the external file inclusion facility of \*GET or \*INCLUDE, source files of any length can be compiled, up to free memory limits in the compiler data tables and loadable machine language file size. Due to the large amount of space available with CED (around 30K), this is unlikely to be a problem. \*GET is usually useful for including standard library subroutines or user functions/commands.

To re-iterate, if during an interactive 'RUN', any errors are detected during compilation, control reverts to the editor at the end of the first pass, with the original source file automatically intact. Otherwise, TEMP/CMD is loaded and executed. When the program is exited (via END or STOP or BREAK) control passes back to the editor, with source text reloaded, unless Z80 code or a compiler bug has caused a serious problem.

### CAUTION

Do not attempt to invoke from DOS Ready, a program compiled from the supervisor mode. To generate a program that is to be invoked from DOS Ready, recompile the source program using BC/CMD.

## 2.2 Runtime Errors

A program will terminate, unless an 'ON ERROR GOTO' is active, when an error condition is detected. If 'ON ERROR GOTO' is inactive, then:

```
RUNTIME ERROR CODE ccc IN SOURCE LINE #xxxxx
```

will appear ('xxxxx' will be invalid if the source line was unnumbered or if the line # information was suppressed in the compiled code with the 'NS' directive).

If compilation was invoked from an interactive RUN, control will be passed back to CED and the source reloaded. If general compilation was used (described in the following section), control will pass back to DOS READY.

A complete list of runtime errors is given in chapter 8. Note that certain special DOS error codes, different from standard or unique codes, will be flagged by being in the range 32-100, with 32 added to the original code to produce the EnhComp code. The DOS error code must be between 0 and 68 to avoid confusion with other EnhComp error codes.

### 2.3 Transferring EnhComp to TRSDOS 1.3

The following procedure is used to transfer your EnhComp system to a TRSDOS 1.3 disk. Note that the procedure requires a two-drive system.

- 1) Place a blank diskette in drive 1 and a working backup of EnhComp in drive 0. BOOT the EnhComp disk.
- 2) Type: `FORMAT :1 (NAME="ENHCOMP",SDEN,CYL=35,Q=N,ABS)`
- 3) After the disk format successfully completes, type: `COPY BC/CMD:0 :1`
- 4) Continue to copy from drive 0 to drive 1 (as you did in step 3) the files: `S/CMD`, `CED/CMD`, `REF/CMD`, and `SUPPORT/DAT`.
- 5) Remove the EnhComp system disk from drive 0 and BOOT your TRSDOS 1.3 system disk. Your TRSDOS 1.3 system disk should have at least 102 granules of free space.
- 6) Use TRSDOS 1.3's CONVERT utility to transfer the five EnhComp files from drive :1 to drive :0.
- 7) EnhComp should now be accessible to TRSDOS 1.3.

### 2.4 General Compilation Parameters

The general format of a direct compiler invocation is:

**BC filespec,start\_address,top\_address,-dir-dir...**

Filespec	is the source program specification. The extension defaults to '/BAS'.
start_address	is the specified program origin.
top_address	is the highest address to be used by the compiled program.
-dir	is a compiler directive.

As you can see, a number of variables can be changed in the invocation. The default loading address for compiled programs is 5200H (Model I/III) or 2600H (TRSDOS 6). You can change this by simply putting a comma after the filespec, followed by the desired address (in hexadecimal format). If it is necessary to limit the top memory location accessed by the compiled program, this limit can be specified as well (for example, to limit access in a 32K RAM program, BFFF would be given, the topmost valid memory location in a machine with 32K of memory). The default top\_address used would be that recovered from the system's HIGH\$ memory pointer at the time the compiled program was invoked.

You can change compilation parameters through a device known as "directives" -- so called because they are directions to the compiler, not compilable

instructions. Directives produce no code per se, although they may affect the size of the final compiled program. Directives specified in the compiler invocation input are "global" directives, so called because they affect the entire source program. You can also use directives within your source program, in which case they're called "local" directives. Some directives can be used both globally and locally. The rest are restricted to either domain. Local directives are explained further on.

As an example, the 'NO' global directive inhibits the generation of an object file, usually to compile a program to check for errors, without overwriting an existing object file. In the case of the TEST/BAS program, this goes as such:

```
BC TEST/BAS,,, -NO
```

Note the omission of the loading origin and memory limit variables. They retain their default values. However, the commas are necessary to delimit the sentence. 'BC TEST/BAS -NO' is invalid, as is 'BC TEST/BAS, -NO' and 'BC TEST/BAS,,, -NO'.

Multiple directives are delimited by dashes, as in:

```
BC TEST/BAS,8000,F000,-WD-WE
```

In addition to the global compiler directives, which may be used, in most cases, both globally and locally, there are purely local directives, which are prefixed by an asterisk (except for Z80-MODE and HIGH-MODE). This is indicated in the directive list, which follows. Note: It is important to realize that compiler directives are activated as they are encountered in the input stream in a purely linear manner from left to right; runtime program logic has no effect on their activation. Directives valid both locally and globally are prefixed with an "\*-"; directives valid only within the program (locally) are prefixed with only "\*".

## 2.5 Compiler Directives

BC supports the following compiler directives:

```
GET, INCLUDE, LIST, PRT, NOLIST, NOPRT, WD, NO, Z80, NS, YS, WE,  
NX, YX, IF, ENDIF, INJECT, LINK, PRINT.
```

In the following paragraphs, directives that are considered global in nature will be denoted with '(G)', while directives that are considered local in nature will be denoted with '(L)'. Directives that are considered both local and global with are denoted with '(B)', while directives that are purely local with '(P)'.

Remember, when you use a compiler directive within your source stream, each must be prefixed with an asterisk and dash ('\*-') except for PURELY LOCAL directives which are prefixed with an asterisk only.

### LIST (B)

This directive will list the source program on the video screen during the second pass, with error messages.

**PRT (B)**

This directive will print the informative and diagnostic messages as well as the source program to your line printer during the second pass, with error messages.

**NOLIST (L)**

This local directive will turn off the source program screen listing until a subsequent LIST directive is detected.

**NOPRT (L)**

This directive will turn off the printer listing until a subsequent PRT directive is detected.

**WD (B)**

This directs EnhComp to write the reference data file upon completion of the compilation phase. The file specification used for the reference file will be constructed with the filename of the source program and the file extension of '/DAT'. No drive extension will be appended. An informative message will be issued advising you of the generation of the file. This file can be subsequently processed by the REF/CMD utility to produce a program reference report.

**NO (B)**

This tells the compiler to refrain from writing the compiled program to a disk file. You will find it useful to speed up the compilation phase when you only want to scan for detectable source code program errors.

**Z80 (G)**

This directive causes the compiler to assume that your source program contains only Z80 assembly language. The compiler will then inhibit writing of "extraneous" high-level support code.

**NS (B)**

This directive tells the compiler to inhibit the generation of source line number information in the object code file of the compiled program. This saves 3 bytes per source code line; however, runtime diagnostics will not be able to then report the line number of a source line which causes a runtime error. The compiler default is to generate source line number information.

**YS (L)**

This directive informs the compiler to resume the generation of source line number information (see directive NS).

**WE (B)**

This directive will cause the compiler to wait for you to press a key when an error has been detected during compilation. This allows you to observe the error diagnostic message without worrying about it scrolling off the video screen. Any keystroke will cause a continuation of the compilation.

**NX (B)**

The compiler normally generates code which checks for the BREAK key and handles TRON at the conclusion of each source program statement. If you do not desire this BREAK key handling, the NX directive will inhibit the writing of this code. This will shorten the resulting compiled program file. Note that the local directive 'YX' can resume the generation of this handling code so that you can restrict certain segments of your program from having the BREAK handling code.

**YX (L)**

This directive resumes the generation of the BREAK and TRON handling code. See the 'NX' directive discussion.

**IF exp <lines of source code> ENDIF (P)**

The IF...ENDIF directive pair provides for a conditional compilation. If the expression, 'exp', evaluates to a non-zero value then the next lines of source up to the 'ENDIF' are compiled. Otherwise, a zero value of 'exp' results in the compiler ignoring the next lines of source until the 'ENDIF' statement is reached.

**\*INJECT filename <(offset<,lower\_limit<,high\_limit>>) (P)**

This directive is used to insert a machine language load file into the current compilation machine code output file. If 'offset' is given, the file will be loaded into memory at a new address of 'offset+old address'. To selectively offset program loading -- say, to avoid offsetting a load to addresses in lower RAM -- a 'lower\_limit' can be given (such as 4400H). Similarly, an 'upper\_limit' for the offset can be given. Thus, to offset the loading of TEST/CMD between all addresses in the range 6000H-7000H by 8000H, use:

```
*INJECT TEST/CMD(8000H,6000H,7000H)
```

This instruction would then inject TEST/CMD into the output stream of the compiled program file. The DOS loader will then load TEST/CMD into memory along with the compiled program; any parts of TEST/CMD that would have loaded between 6000-7000 will now load into memory at E000-F000.

**\*LINK filespec(module #, module #, ...) (P)**

This directive causes the compiler to link a special link-type file into the current compiled program output. Such a file would be provided and its use documented by the publisher of EnhComp. The SUPPORT/DAT library file is an example of such a link file. In addition to greater disk space efficiency, link files are "assembled" much faster than the original source.

**\*GET/\*INCLUDE filespec (P)**

The two directives 'GET' and 'INCLUDE' are equivalent. They are used to include a secondary source program file into the input stream. This can be useful to provide a means of segregating your source program into "modules" - each module in a separate file. At the conclusion of the 'INCLUDE' file, the source stream compilation will revert to original source program.

**\*PRINT<#n> <"info"> <,> <;> <\$(chrexpr)> <exp> (P)**

This directive is used to display a compilation message on the screen or printed on a printer, depending on the current option switch settings. The '#n' specifies the pass in which to print (if omitted, the second pass only is implied). If '#n' is entered as '#0', then the message will print during both passes. A '#1' or a '#2' entry indicate that the message will print only on the first or second pass respectively. Anything in quotes is printed verbatim. The '<,>' and '<;>' are print delimiters as in a normal BASIC PRINT statement. For an entry of '\$(chrexpr)', the equivalent ASCII code is printed. The field denoted as 'exp' indicates a print expression.

## **2.6 Compilation mode versus Interactive RUN mode**

The interactive RUN mode is useful for writing and debugging programs. The /CMD file produced during this time, TEMP/CMD, is not intended to be used without the S/CMD supervisor loaded and CED/CMD available on the disk.

To produce a final, compiled program once development is complete, you must invoke BC/CMD directly from DOS level. The various optional parameters or directives available have been described in the last section. It might be desirable to disable the "debugging friendly" features in the compiled program (source line # printed on error, BREAK detected, TRON available) for your final copy; in addition to saving space, this will make it impossible for someone to decode your program without a lot of work.

This program will be in the form of a fully independent '/CMD' file, executable as easily as an other /CMD file. BC/CMD, S/CMD, CED/CMD, and SUPPORT/DAT will no longer be needed to run the program.

## **2.7 Independent use of compiled programs**

There are no restrictions (royalty payments) on compiled programs to be distributed for NON SYSTEMS SOFTWARE or UTILITIES use, such as a business program. For SYSTEMS SOFTWARE/UTILITIES (such as another compiler, or a language, and so on -- in general, anything designed to be a programming tool), public distribution is PROHIBITED without a written release from the author of EnhComp (Philip Oliver), or some kind of fee-per-copy arrangement. Without such a release or arrangement, such distribution will be considered copyright infringement of the SUPPORT/DAT subroutines.

### 3 CED/CMD Editor

#### 3.1 CED General Information

The EnhComp editor differs somewhat from the TRS80 BASIC editor. However, internal editing commands (with the 'E' command) are the same. The significant difference between the TRS-80 BASIC editor and the EnhComp editor is that the latter recognizes two types of line numbers: editing line numbers, and BASIC line numbers.

Any individual line may carry a distinctive line number, treated as a BASIC line number; for this reason, standard ASCII BASIC programs can be loaded into the EnhComp editor. Every line is numbered from 1 thru "n" in steps of one; also, where "n" is the total number of program lines. Not every line has to have a BASIC line number, but with every line is associated an edit number, representing its position relative to the beginning. The advantage of this is never having to renumber due to the line numbers being too close together. The disadvantage lies in the fact that "renumbering" occurs automatically whenever you insert, delete, copy or move lines. You must therefore keep track of where you are in the program.

If multiple (edit) line number expressions are needed by a command, they are always separated by commas. An edit line number expression can consist of a decimal number, or the letter "T" to represent "1" (the top), or "B" to reference the bottom (last) line. Note that 'DET,B' deletes your entire program (DElete from Top to Bottom.)

To recover from an unforeseen accident during a compiled program run, recall that your source text is always saved in "TEMP/BAS" if compilation was invoked from edit mode. All you have to do is reload it.

NOTE: Unless otherwise mentioned or clearly implied by the context, references to line numbers are EDITOR line numbers.

#### ? exp

This command will print the integer result of the expression, 'exp'.

#### ?F

This command will print the filename of the file currently being edited.

#### / editor\_line\_number, BASIC\_line\_number

This command will add the specified BASIC line number to the line identified by the given editor line number.

#### < BASIC line number

This command will remove the specified BASIC line number from whatever editor line it is on (if it exists).

#### BLH

The "BASIC Line Hide" command will suppress the display of all BASIC line numbers.

### **BLS**

The "BASIC Line Show" command will restore the display of BASIC line numbers.

### **C start\_line,end\_line,destination\_line**

This command will copy a block of lines from the 'start\_line' to the 'end\_line' (inclusive), inserting at the 'destination\_line'.

### **DE line1 <,line2> (DL ... for BASIC line #s)**

The 'DE' command will delete a single line identified by 'line1'; or the multiple lines identified by 'line1' through 'line2', if 'line2' is given. Using 'DE', the line numbers entered for the deletion refer to EDITOR line numbering. If you wish to delete a line or lines according to their BASIC line number(s), specify the delete command as 'DL' instead of 'DE'.

### **ELH**

The "Editor Line Hide" command will suppress the display of EDITOR line numbers. This is the default mode of CED.

### **ELS**

The "Editor Line Show" command will restore the display of EDITOR line numbers.

### **ERROR errcode (or ERR errcode)**

This command will display the full error message of the given runtime code denoted by 'errcode'.

### **Fstring**

Beginning at the current line+1, this command searches through the text for the specified string. The line which contains the string is listed if a match is found, otherwise 'STRING NOT FOUND' is issued and the search stops. IMPORTANT NOTE: Do NOT include ANY SPACES after the 'F' command unless they are part of the search string.

### **GO**

This command causes an exit from the editor and a return to DOS.

### **E or EDIT <'string'> <linerange> (use "ED" for editor line #'s)**

The 'E' command is the most sophisticated of the edit commands, not surprisingly. It allows intra-line editing of a particular line or set of lines on a mostly conceptual basis (as opposed to directly perceptual screen editing.) Users of Z80 based TRS-80 computers will recognize the format of the command, since it is essentially the same as the EDIT function of the BASIC language on those computers.

Note that with the 'E' (or 'EDIT') command, numbers refer to BASIC line numbers; with the 'ED' command, numbers refer to editor line numbers. Otherwise, all material in this description is precisely the same for both commands.

Fundamentally, editing is done by single letters, which switch the editing mode when appropriate. Initially, only the line number is shown; the cursor is placed at the beginning of the line. This is the edit command mode.

**Summary of internal edit commands|**

<space>	Skip over next character, displaying it
<backspace>(edit)	In edit mode: Move cursor left nondestructively
<backspace>(ins)	In insert mode: Move cursor left destructively
A	Leave the edit with the old line untouched
C<char>	Change characters
D	Delete character
H	Hack line
I	Go into insert mode
K<char>	Delete up to <char>
L	List rest of line and restart edit on new line
S<char>	Move cursor to occurrence of <char> after cursor
X	Move cursor to end of line, start insert mode

To non-destructively move the cursor over the line and to display it one character at a time, press the space bar. The cursor won't move past the end of the line once the last character has been displayed. To non-destructively move the cursor backwards, press the backspace key. Once again, once the first character has been moved over, the cursor won't move. The space and the backspace can be seen as single letter commands.

To list the entire line and then restart the edit at the beginning of a new line, type 'L'. Doing this twice will show you a "clean" version of the line you're working with.

To insert new characters into the line, position the cursor to the desired point (directly over the point of insertion) and type 'I'. Then, any characters typed will be inserted into the line at that point; what you see from the line number on will be the start of the new line. Any backspaces in insertion mode are destructive. To stop the insertion and go back to the edit command mode (the initial mode), press the <ESC> key (or <SHIFT-UP-ARROW>).

To delete characters, position the cursor directly before the character to be deleted and type 'D' (in the edit command mode.) The character just deleted will be printed between slash marks.

To totally restart the edit from scratch, and call up the line as it was initially before your editing, type 'A' in edit command mode. The edit will be restarted on the next line.

To "hack" the rest of the line at any given point, type 'H'. The cursor will then be placed at the end of the line and insert mode will be on.

To change a character "under" the current cursor position, type <C><char>; the character will be changed to <char>.

To delete all characters from the character "under" the cursor up to and including a particular character, type <K><char>.

To move the cursor to the end of the line and go into insert mode, type <X>.

To move the cursor to a particular character in the line after the cursor position, type <S><char>. If the specified character is not on the line, the cursor will be moved to the end of the line. If it is, the cursor will be placed "over" that character. In either case, edit command mode will still be active.

Note that pressing the <ESC> key or its equivalent <SHIFT-UP-ARROW> will almost always abort the current command and cause a return to edit command mode.

Once all editing has been completed and you're satisfied with the results, hitting <ENTER> will enter the new line in place of the old one. If you want to leave the line alone, type <A> in edit command mode followed by <ENTER>; the line will be unchanged. Hitting <BREAK> will also cause an escape without changing the old line.

Optionally, you can initially specify two parameters. If you specify a range of lines, a succession of edits will occur. In this case, after you type <ENTER> or <A> to enter or escape from the edit, the next line will be edited. However, typing <BREAK> will cause a return to the editor command mode.

You can also specify a string which will be entered just as if you had typed it in at the beginning of the edit. For example, entering:

```
E'L'10
```

would edit line 10, displaying it first, because of the <L> edit command. Note that the apostrophes are actual characters to be typed, not documentation syntax marks.

This is really only useful when a range of lines is specified. Then, you can automatically edit them without tediously typing the edit commands for each line. A left bracket, "[", in the string is taken to mean an <ENTER>, so entering, for example:

```
E'I;['15,20
```

would insert a semi-colon at the beginning of lines 15 through 20 inclusive, editing each line automatically. This particular command would be useful to temporarily convert a range of Z80 assembler source lines to comments. Later, the semi-colons could just as easily be deleted by entering:

```
E'D['15,20
```

Note that if the parameter "T,B" (without quotation marks) is specified for the line range, the entire program will be edited.

As alluded to earlier, typing a number before most commands will cause that command's action to be done that number of times. For example, typing <1><2><space> essentially causes the space command to be done twelve times. If the end of the line isn't reached, the cursor will skip over twelve new characters. To delete 6 characters, say, type <6><D>. To "erase" a number just typed and essentially set it back to 1, type <ESC> or its equivalent.

With the <S> and <K> commands, the specified number of characters will be searched before the command's action is done. For example, <2><S><A> will skip the cursor over the first 'A' encountered in the line and place it over the second one found (or the end of line, whichever comes first). In addition, <3><K><I> will delete all characters from the one "under" the cursor to the third 'I' found in the line after the cursor, inclusively -- or until the end of the line is reached.

With the <C> command, the specified number of characters will be modified. If the end of the line is reached, edit command mode is enabled.

#### **H line1 <,line2>**

This command will print 'line1' (through 'line2' if given) on your printer. If the printer is unavailable, hit <BREAK> to escape.

#### **I line\_number**

This command will begin insertion of lines at the specified line number. Hit <BREAK> to escape insert mode. Note that no BASIC line number is attached to these lines.

#### **K:filespec**

This command will "Kill" (remove) a file from disk. Note the use of the mandatory colon, ':', in the command syntax.

#### **LIST linerange**

This command will list a range of lines to the video screen; numbers given by 'linerange' refer to BASIC line numbers.

#### **LLIST linerange**

This command will print a range of lines on your printer; numbers given by 'linerange' refer to BASIC line numbers.

#### **L:(insert line)> filespec <,line1 <,line2>>**

This command will load source text from disk into memory. Note the use of the mandatory colon, ':', in the command's syntax. Note also that line numbers are EDITOR line numbers. The simplest form of this load command is, for example:

```
"L:TEMP/BAS"
```

TEMP/BAS will be either loaded into memory if there's nothing in the text buffer, or appended onto the end of the current text.

If "(insert line)" is specified, the disk file will be inserted into that point in the current text.

If <line1, <,line2>> is/are given, only 'line1', or 'line1 through line2' inclusive, is/are loaded from the disk file (relative line numbering is used). For example:

```
L:(10)SOURCE1/BAS
```

Inserts "SOURCE1/BAS" starting at line 10.

```
L:CHESS80/BAS,50,177
```

Loads or appends lines 50 through 177 from the "CHESS80/BAS" file. Loading stops automatically if less than 177 lines are in the file.

```
L:(184)NWAR/BAS,15,40
```

This is a combination of insert/selective loading. Lines 15-40 from "NWAR/BAS" are inserted at the current line number 184.

#### **M line1,line2,destination\_line**

This command is similar to "C"opy, except that lines are moved rather than duplicated.

#### **N <lower\_lim<,upper\_lim<,start<,inc>>>>**

This command renumbers the BASIC lines of a program. Four optional parameters are allowed. The first two are the current line range to renumber. The third is the new starting number. The last is the line increment. The default values are 0,65535,100,10. For example:

```
N 100,300,10,10
```

would renumber all lines in the range 100-300 inclusive; the first line then being 10, the next 20, etc.

```
N ,,100,5
```

would renumber the whole program, starting at 100 and advancing in increments of 5.

#### **O**

This command will begin appending lines without BASIC line numbers.

#### **P line1<,line2>**

'P' lists 'line1' or 'line1 through line2' to the screen. If no parameters are given, then 15/23 lines starting with the current line are listed.

#### **Q drivenum**

This command will display a directory of files on the disk drive specified as 'drivenum'. If 'drivenum' omitted, drive 0 is assumed.

#### **R line1<,line2>**

'R' will replace 'line1' or 'line1 through line2'. The current line is printed; insert prompt allows new replacement line to be entered. Once line(s) are replaced, control passes automatically into insert mode.

## **RUN**

This command starts a chain of events if the compiler editor is invoked in the supervisor mode (i.e. from "S/CMD"). First, source text is saved in the file named, "TEMP/BAS". Then it's compiled into "TEMP/CMD". If the compilation is successful, "TEMP/CMD" is invoked; if not, control passes to the editor, with source reloaded. This also happens when the runtime program terminates in an acceptable (END/STOP/BREAK) way.

## **Sstring**

This command operates the same as 'Fstring' except the search starts at the beginning of the text instead of line+1.

## **U**

This command provides memory usage. It displays number of bytes used and bytes free.

## **V:filespec <line1<,line2>>**

This command allows you to display lines from the specified disk source text file.

## **W:filespec <line1<,line2>>**

This command writes text from memory to the specified disk file. Note the use of the mandatory colon, ':', identified in the command's syntax. If line parameters are omitted, the entire text is saved. If line parameters are given, only those lines are written to the file.

## **X/replacement\$/search\$**

This command will search and replace all occurrences of the search\$ string with the replacement\$ string. The search will begin at the current line number. A <BREAK> stops the command. Note that only one replacement per line is done. For example:

```
X/ent/ant
```

replace all occurrences of "ant" with "ent".

## **Y=linespages,pagelength**

This command will change printer forms control parameters (for LLIST, H) to do a top\_of\_form, 'TOF', after 'linespages' lines. If 'pagelength' is given, this will define the number of lines total for each page of the paper you're using in your printer (usually 66).

## 4 EnhComp BASIC Statements and Functions

### 4.1 Compiler Introduction

EnhComp is a compiler, which differentiates it from TRS-80 BASIC interpreters included with your DOS. The essential difference is not so much the structure of the languages themselves, but the manner in which your computer executes any given program in the languages. The resident BASIC in your machine must analyze program text every time it executes a command. Compilers, however, translate program text into a format that is better suited to machine interpretation than a straight BASIC program.

Some compilers compile to "pseudo-code", which is space efficient but slow. EnhComp is a true compiler; it compiles directly to Z80 machine language. EnhComp does accomplish some space compression since many lengthy routines used many times throughout compiled programs are copied just once in memory, and called as subroutines.

EnhComp is unique. Not only can the programmer take advantage of a powerful high level language, but Z80 source code can be intermixed with the language to any extent desired. In fact, EnhComp is not only a compiler, but a Z80 assembler that allows powerful algebraic expressions in source code statements. It takes advantage of the high level language/machine language intermix ability, with special functions that allow access to variable, line number, and label addresses.

EnhComp is not guaranteed to translate your TRS-80 Model I or III interpreted BASIC programs unmodified into machine language. However, any differences are slight and easily fixed to accomodate compilation. The large repertoire of new commands and functions make it likely that you will be writing old programs over using these new features, rather than settling for the limited capabilities of the resident BASIC/Disk BASIC interpreters.

EnhComp retains many of the "nice" features of interpreted BASIC that are excluded in other, inferior, compilers. For example, the <BREAK> key is functional during execution, if desired, and BREAKing a compiled program will result in a BREAK message along with the source code line number in which the interrupt occurred. Error messages at runtime display the error code and the source code line number in which the error occurred.

Dynamic array allocation with up to fifteen dimensions (A(a1,a2,a3,...,a15)), is allowed, as is dynamic string space allocation. All standard BASIC variable types are supported (integer, single precision, double precision, and string). Strings are no longer limited to 255 characters in length; 32767 is the new string length limit. "FOR-NEXT" constructs may have more than one NEXT for a single FOR, since error checking (in this case) is done at runtime, not at compile time. More than one dimension statement for the same array may occur in a program at once, but an error message will be issued at runtime if more than one of the dimensions are executed.

### 4.2 Compiler Directives

Compiler directives are not "true" commands. They simply tell the compiler, at compile time, to do some task. The directives pertinent to the program code stream will be discussed here. All of the compiler directives are discussed in Chapter 2.

## **HIGH-MODE**

This puts the compiler into High Level Compilation mode. 'HIGH-MODE' is the default compilation mode. The compiler will be looking for only "high level" commands and functions in this mode.

## **Z80-MODE**

This puts the compiler into Z80 Assembler mode. High Level commands will generate expression errors in this mode. Only valid Z80 opcodes and assembler directives will be recognized. Source code line inclusion and BREAK key checking will be disabled in this mode.

## **High Level Statements**

Statements are instructions that perform some specific task, and exist as independent entities; as opposed to functions, which are used inside algebraic or string expressions, and are not used independently. Statements and functions may be used in High Level mode only (the default mode of the compiler.) They will generate expression errors in Z80 mode.

## **High Level Functions**

Functions are used with expressions. They are also used with statements; however, a function is never used alone. In general, functions can be divided into two main categories: String and Numeric. Naturally, these categories are further divided into fairly reasonable groups of related functions.

### **String Functions: An overview**

Strings, as you're probably aware, are bytes which are sequentially strung together in a "string" and which can be assigned and manipulated using string variables, which can hold a string of variable length. With EnhComp, this length can be from 0 to 32767, a significant improvement over the 255 character limitation of many interpretive BASICs.

EnhComp internally uses a memory-efficient string list technique to manipulate strings. This process is transparent to the user; it is worth mentioning because PRINTs or LPRINTs take up no extra string space whatever when printing a string expression -- except a small amount for generative string functions such as HEX\$ and BIN\$. Additionally, string assignments are fairly memory and time efficient due to the fact that string literals and STRING\$ functions take up no temporary string space during the assignment; however, A\$=A\$+B\$, say, requires that A\$ and B\$ take up temporary storage space due to extensive moving around of A\$ and B\$ during the assignment.

However, the same expression, A\$+B\$, would take up NO temporary space if it was printed (PRINT A\$+B\$ or LPRINT A\$+B\$), regardless of the combined length of A\$ and B\$. In the same way, LPRINT "--> "+STRING\$(128,42)+" <--" would work with 0 bytes cleared for string space.

## **4.3 Function Reference**

The following pages list all the built in statements and functions

This function returns the absolute value of its argument.

ABS(exp)	FUNCTION
exp	- is a numeric expression.

ABS returns the absolute value of an expression. If the expression evaluates to a non-negative value, that result is returned; otherwise -expression. For example: ABS(-4) = 4; ABS(0) = 0; ABS(1.414) = 1.414.

This function obtains the absolute memory address of its argument.

ADDRA(addr)	FUNCTION
addr - is a line number or a label.	

ADDRA returns the absolute memory address of a specified line number or label. For example:

```
10 L=ADDRA(100)
20 A=PEEK(L):L=L+1:IF A=0 THEN END
30 PRINT CHR$(A);:GOTO 20
50 Z80-MODE
100 "STRING":DB 'ASCII TEXT STRING',13,0
```

This prints a string defined in memory, accessible as the address of line numbered 100. Alternatively, line 10 could be: L=ADDRA("STRING"), as the value of the label "STRING" equates to ADDRA(100).

This is used to allocate the quantity of disk file control blocks.

ALLOCATE exp	STATEMENT
exp	- is the number of file control blocks to allocate in the range <1-15>.

Before any disk files can be OPENed, file control blocks must be allocated. 'ALLOCATE' creates up to 15 control blocks. Note that the blocks are allocated sequentially -- blocks allocated equal the highest file buffer accessible by OPEN.

For example, if a maximum of 3 files will be open at once in a program, 'ALLOCATE 3' is executed before any OPENs are done.

File control blocks can be specified by a variable expression - the number of blocks to be allocated needs not be a constant defined at compile time. For instance, ALLOCATE F+1 is valid.

More than one ALLOCATE can exist in a program -- but only one of them may be executed (or an error will be generated.)

These functions indicate their arguments as being other than decimal format numbers.

&Bd0...d15	- Binary number	FUNCTION
&Hd0...d4	- Hexadecimal number	FUNCTION
&Od0...d5	- Octal number	FUNCTION

'&B' signals a binary number in ASCII format. For example, the assignments: "A = B AND &B11110101" and "A = B AND 245" are functionally equivalent. '&H' flags a hexadecimal ASCII format number: &H100 = 256 decimal; and '&O' flags an octal ASCII format number: &O70 = 56 decimal.

This function returns the first byte of its string argument as an integer.

ASC(exp\$)	FUNCTION
exp\$ - is any string expression.	

'ASC' takes the first byte of the specified string expression and converts it into numeric format. For example:

```
10 A$="ABC"  
20 PRINT ASC(A$)
```

prints 65, the ASCII code of the letter 'A', which is the first character in the argument, A\$.

This function obtains the arc tangent of its argument.

ATN(exp)	FUNCTION
exp	- is a numeric expression in radian measure

ATN returns the arctangent of an angle assumed to be in radian degree measure. It can receive, and return, either a single or a double precision value, of full precision. Thus, if the argument is double precision, the result will be a double precision value.

This function converts numeric expressions to a string of binary digits.

<pre>BIN\$(exp16)                                FUNCTION exp16    - is in the range &lt;-32768 to 32767&gt;</pre>
--

BIN\$ returns a 16 character ASCII binary representation of a selected integer expression. For example, BIN\$(4095) is equal to "0000111111111111".

These statements are used to provide <BREAK> key control of your program.

BKON	STATEMENT
BKOFF	STATEMENT

'BKON' and 'BKOFF' can be used to effectively turn the BREAK key on or off, respectively. They affect only the BREAK scan flag. BKON will have no apparent effect if the "-NX" directive flag has been specified, since the BREAK scan code calls will be left out of the compiled program.

An 'ON BREAK GOTO addr' causes a jump to the specified line number or label if the <BREAK> key is hit and the BREAK scan is activated. 'ON BREAK GOTO 0' disables <BREAK> key branching, parallel to 'ON ERROR GOTO 0'. Causing an 'ON BREAK GOTO addr' jump also automatically disables <BREAK> key branching.

#### Example Program

```
5   ON BREAK GOTO 100
10  PRINT"HO HUM ..."
20  FOR X=0 TO 1E12: NEXT
30  PRINT"OH BOY, LET'S COUNT TO A QUADRILLION NOW!"
40  END
100 PRINT"THANKS! SAVED FROM A FATE WORSE THAN SCARFMAN...."
```

This function is used to convert its argument to double precision.

CDBL(exp)	FUNCTION
exp	- is a numeric expression.

CDBL converts a numeric expression to double precision floating point format.

This function converts a byte value to a one-character string.

CHR\$(exp8)	FUNCTION
exp8	- is in the range <0-255>.

CHR\$ is used to convert a number between 0 and 255 into a string character.  
CHR\$(65) = "A" for example.

This function converts a numeric expression to integer format.

CINT(exp)	FUNCTION
exp - is a numeric expression.	

CINT converts a numeric expression to integer type. Expression must be in the range (-32768 to 32767).

The 'CLEAR' statement is used to clear variables and allocate string space.

<code>CLEAR &lt;exp&gt;</code>	STATEMENT
<code>exp</code>	- is used to designate the amount of string space to reserve.

'CLEAR' without expression simply zeroes all numeric variables, clears all strings, and undimensions all arrays. With expression given, 'CLEAR' does all of the previous and also redefines the amount of memory devoted to string storage, which is 100 bytes by default.

If, for example, you had a program that stored a maximum of 500 strings each with a maximum length of 8 bytes, then you would need to at least CLEAR 4000 (bytes). In reality, string related functions and commands temporarily use some of the currently free string storage area as a "scratchpad", so a buffer of 600 bytes is not unreasonable -- make it: 'CLEAR 4600'.

This statement is used to close a file or files.

```
CLOSE <blknum <,>blknum ...>>          STATEMENT  
  
blknum   - designates a specific file to close. If no  
           blknum is given, all open files are closed.
```

All open files must come to a close. 'CLOSE' assures that all important information vulnerably sitting in RAM is written safely to disk. (Disk data is usually unaffected during "I" type file access so ACCIDENTALLY not closing an "I" type file is usually harmless. CLOSE them anyway.)

With a list of file control blocks given, only those blocks will be affected. CLOSED control blocks are unaffected by CLOSE. With no specific File Control Blocks listed, ALL open files are closed.

This statement is used to clear the video display screen.

CLS	STATEMENT
-----	-----------

This statement simply clears the screen with blanks (ASCII 32) and homes the cursor. Only a portion of the screen will be cleared if scroll protection is enabled.

The COMMAND-ENDCOM construct permits you to define new BASIC commands.

COMMAND name(input variable list)	STATEMENT
program statements	
ENDCOM	STATEMENT
name	- is a string of characters in the set ("A"- "Z", "0"- "9"), starting with ("A"- "Z")
input variable list	- is a list of (local) input variables.
Note:	user commands are invoked by preceding the name with a percent as in, '%name(operand list).

COMMAND is a powerful statement that allows you to define new commands. A user-command definition consists of the 'COMMAND' statement header, a definition body, and an 'ENDCOM' statement. Once defined, the user-command is easily and clearly referenced by the technique of "%name(operand list)". The percent sign acts as a user-command invocation symbol.

Any combination of numeric and string expressions can be specified as user-command operands. For each operand specified in a user-command invocation there must be a corresponding local variable in the user-command definition - "local" because the existing values of the variables listed in the definition are pushed onto the stack before they are assigned to the operands given in the user-command invocation. NOTE: input variables are restricted to simple variables and exclude array elements. So ALPH\$ is a valid local input variable, but NAME\$(4) is NOT.

The RETURN command (inside a user-command definition), re-assigns original values to local variables and exits from the user-command.

COMMAND definitions may not be nested. Also note that definitions are "defined" at compile-time, so they may exist anywhere in the program; they need not be executed. In fact, when encountered, a definition is skipped over.

Example Program #1:

```
10 PRINT"FACTORIAL PROGRAM":PRINT
20 INPUT"# TO TAKE FACTORIAL OF";X
30 IF X<>INT(X) OR X<0 THEN PRINT"INVALID #.":GOTO 20
40 %FACTORIAL(X):PRINT X;"! = ";F
50 END
60 COMMAND FACTORIAL(Y)
70 IF Y<2 THEN F=1:RETURN
80 %FACTORIAL(Y-1):F=Y*F:RETURN
90 ENDCOM
```

The preceding program needs a little explaining. The command definition body, lines 70-80, is the heart of the program. Line 70 sets 'F', the output variable by choice, to 1 if 'Y', the local input variable is less than 2; as it should, as 1! = 0! = 1. Line 80 is the clincher. %FACTORIAL(Y-1) is a recursive invocation, so called because the user-command definition is

referencing itself! The opinion of poor math teachers aside, definitions that refer to themselves can be perfectly valid (with the important proviso that at some point something specific must happen and the recursion, or self-referencing, terminates); in this case %FACTORIAL(Y-1) is allowable because of the fact that 'Y' is a local variable. Intermediate values in the factorial calculation are preserved.  $F=Y*F$  is a perfectly proper way to calculate the factorial, because  $Y! = Y * (Y-1)!$ , and F (before the assignment  $F=Y*F$ ) is  $(Y-1)!$  because of %FACTORIAL(Y-1). Naturally, a recursive invocation has to stop sometime for it to be useful, and the "stopper" is line 70, which returns a "hard" number (1) when Y is finally decremented to 1. From then on, a sort of backlash occurs until the factorial is finally calculated. Details are left "... as an exercise for the reader."

The potential power of mixing Z80 assembly language with BASIC should be evident in the next program.

Example Program #2 for TRS-80 Model I/III:

```
10  FOR X=0 TO 255
20  %FILL(X)
30  NEXT
40  END
45  '
50  COMMAND FILL(X%)
60  Z80-MODE
70  LD A,(&(X%)):LD HL,3C00H:LD (HL),A
80  LD DE,3C01H:LD BC,03FFH:LDIR
90  HIGH-MODE
100 ENDCOM
```

Screen memory is filled with all possible characters, making a rapidly changing display. You Z80 programmers can figure this program out. The rest of you -- what can I say? ('learn Z80 assembly language ...').

This statement is used to complement a pixel.

COMPL(x,y)	STATEMENT
x	- is a numeric expression which evaluates to the range <0 - 127> for 64-column screens and <0 - 159> for 80-column screens.
y	- is a numeric expression which evaluates to the range <0 - 47> for 16-row screens and <0 - 71> for 24-row screens.

SET, RESET, and COMPL form the set of the single-pixel-affecting graphics commands. Note that screens that display 16 rows of 64 characters will display 72 rows by 160 columns of graphics pixels; screens that display 24 rows of 80 characters will display 72 rows by 160 columns of graphics pixels.

The COMPL command complements a selected graphics pixel, turning it ON if it is OFF and vice versa. The following illustrates a brief example of these graphics commands:

```
5   Y=23:RANDOM:CLS
10  FOR X=0 TO 127
20  SET(X,Y)
30  Y=Y+SGN(RND(3)-2)
40  IF Y<0 THEN Y=0 ELSE IF Y>47 THEN Y=47
50  NEXT
60  FOR X=0 TO 127
70  COMPL(X,23):NEXT
80  FOR X=0 TO 127
90  RESET(X,23):NEXT
```

The program first plots a pseudo-"mountainous" profile on the screen, proceeds to "complement" all graphics dots down the middle of the screen, and finally resets all pixels through the middle of the screen.

This function obtains the trigonometric cosine of its argument.

COS(exp)	FUNCTION
exp	- is a numeric expression in radian measure.

COS takes the cosine, in radians, of an expression. It returns, in full precision, a value of the same type as exp. Thus, if the argument is a double precision type, the value returned is in double precision with full significance.

This function converts its argument to single precision.

CSNG(exp)	FUNCTION
exp - is a numeric expression.	

CSNG converts any numeric expression of any numeric type into a single precision format number.

This function obtains the position cursor of the video cursor.

CURLOC	- No operands are required!	FUNCTION
--------	-----------------------------	----------

'CURLOC' returns the position of the video screen cursor. The position obtained is a value from 0 to n where n+1 represents the total number of characters displayable on the video screen (0-1023 for 16x64 and 0-1919 for 24x80). 'PRINT @ CURLOC,...' is normally equivalent to 'PRINT ...'.

The 'CVD' function unpacks the 8-byte string argument to a double precision floating point number.

```
CVD(exp$)
exp$      - is an 8 byte string expression
```

CVD's primary purpose is to convert a double precision number stored in a file on disk as an 8-byte string back into double precision format. It is the converse of the MKD\$(exp) string function. MKD\$, described elsewhere, converts a double precision numeric expression into an eight byte string containing the double precision data. EXP = CVD(MKD\$(EXP)).

The 'CVI' function unpacks the 2-byte string argument to an integer number.

```
CVI(exp$)
exp$      - is a 2-byte string expression.
```

The main purpose of CVI is to convert an integer stored as a 2-byte string on disk by the converse string function MKI\$(exp) back into an integer. EXP = CVI(MKI\$(EXP)).

The 'CVS' function unpacks the 4-byte string argument to a single precision floating point number.

```
CVS(exp$)
exp$      - is a 4 byte string expression.
```

The prime function of CVS is to convert a single precision number, converted into a 4 byte string by the MKS\$ string function and stored in a disk file, back into a single precision number. EXP = CVS(MKS\$(EXP)).

This function returns the system date as a string.

DATE\$	There is no operand	FUNCTION
--------	---------------------	----------

The system date is returned as an eight-character string of the form,  
MM/DD/YY.

This statement allows you to declare a list of data items to be input with the READ statement.

DATA datalist	STATEMENT
datalist - is a list of numbers or alphanumeric strings, quoted or unquoted; each item is separated by a comma.	

DATA provides an efficient way to store many static pieces of data in a program (such as a tax table). Executing a DATA statement does nothing as program execution jumps over the data list.

READ is the mechanism used to read from DATA lists. READ has the peculiar attribute that it can read a DATA item as either a string or a number. An item can always be read into a string (as a string of characters). An item can SOMETIMES be read as a number -- if it's a number. READ A\$ reads the next DATA item (say 1.618033) literally, character by character, into A\$; in this case an 8 byte string. READ A, using the same item, stores into A the binary equivalent of the converted string 1.618033.

RESTORE and RDGOTO provide ways to point at the desired data list. RDGOTO, especially, eliminates the wasteful process of reading and discarding lists of data to get to the desired list required in interpretive BASIC.

Initially, the first data item read, unless the data pointer is changed by a RDGOTO/RDGTO statement, will be the first data item in the first DATA statement in the program.

Example Program:

```
5   RDGOTO "PRIME"
10  READ TITLE$:PRINT TITLE$:PRINT:READ N
20  FOR X=1 TO N:READ A:?A,:NEXT
30  END
35  '
40  "FIB"
50  DATA The first EIGHT Fibonacci numbers in order
60  DATA 8, 1,1,2,3,5,8,13,21
70  "PRIME"
80  DATA The first NINE prime numbers in sequential order
90  DATA 9, 2,3,5,7,11,13,17,19,23
```

'DEC' is used to decrement an integer variable.

DEC intvar	STATEMENT
intvar - is either an integer variable or an integer array element.	

'INC' and 'DEC' provide a very quick way to increment or decrement a specified integer variable, respectively.

Examples:

```
INC A%:      'A% = A% + 1
DEC B%(10):  'B%(10) = B%(10) - 1
```

These 'DEFxxx' commands are used to declare a group of variables to be of a specific type: integer, single precision, double precision, or string.

DEFDBL letters	STATEMENT
DEFINT letters	STATEMENT
DEFSNG letters	STATEMENT
DEFSTR letters	STATEMENT
letters	- is a list of letters (A-Z) flagging all variables beginning with specified letter. Multiple letters are separated by a comma in the list. Two letters separated by a dash indicates both letters and all letters alphabetically between them (e.g. B-E specifies B,C,D, and E).

The standard default type for variables, when no type declaration character suffix follows a variable (% = integer with 2 bytes of storage needed, ! = single precision with 4 bytes of storage needed, \$ = string with 4 bytes of storage needed, # = double precision with 8 bytes of storage needed), is single precision. However, the above listed commands alter the default types for selected variables -- all variables beginning with the specified letter(s) in the list. For example, 'DEFINT A-K' instructs the compiler to assume that all following untyped variables starting with one of the letters A,B,C,D,E,F,G,H or K are integers (integer type).

```
*****
*
*   IMPORTANT INCOMPATIBILITY NOTE: All above statements
*   are, in reality, COMPILER PSEUDO-OPs! They affect
*   compiled output as they are LINEARLY encountered
*   sequentially in a source line, not as they are
*   LOGICALLY encountered. For example:
*
*   IF A=3 THEN DEFSTR A-Z ELSE DEFINT A-Z
*
*   sets all following untyped variables to be strings,
*   and then immediately assumes them to be integers.
*   In other words, RUNTIME LOGIC has ABSOLUTELY NOTHING
*   TO DO with setting untyped variable type defaults,
*   unlike interpretive BASIC. In fact, the compiler
*   generates no code for DEFINT,DEFSTR,DEFDBL or DEFSNG.
*
*****
```

This statement is used to define single-line user-defined functions.

```
DEFFN name(input variable list) = exp      STATEMENT  
input variable - is any simple string or numeric  
                variable. Arrays are not allowed.
```

DEFFN is used to define a function capable of being evaluated from a single expression. It operates similarly to Interpretive BASIC. EnhComp uses FUNCTION as a powerful statement that allows new multi-lined functions to be defined.

DEFFN is an interpretive Disk BASIC feature. The statement:

```
DEFFN name(input variable list) = exp
```

is functionally equivalent to:

```
FUNCTION name(input variable list): RETURN exp: ENDFUNC
```

The 'DIM' statement is used to allocate space for one or more arrays while specifying the array dimensions.

<pre>DIM array(explist) &lt;,array2(exp)...&gt;          STATEMENT</pre> <p>array      - is an array name.</p> <p>explist   - is an expression or list of expressions,              specifying the index limits of the array.</p>
---

Until an array is DIMensioned, it cannot be accessed. DIMensioning sets up the index limits (defining the acceptable range of index values) and allocates memory for array data. For example:

```
10  DIM A(10)
20  FOR X=0 TO 11:A(X)=X*X:NEXT
```

will cause an error when X=11, which exceeds the dimensioned limit of 10.

Multiple dimensions can be done with one 'DIM' statement by separating the arrays by commas -- i.e. DIM X(60),Y(75).

EnhComp allows the actual index limits in the 'DIM' statement to be undefined at compile time (in other words, specified by variables and resolvable only at run-time), unlike many other BASIC compilers. For example, the statement:

```
DIM TAX(A,B)
```

is allowed by EnhComp, because the dimension will occur dynamically when the compiled program is run, but disallowed by BASIC compilers that need constants as index limits to precompute the amount of space needed for all array elements.

This statement is used to scroll the video screen down one line.

DOWN	STATEMENT
------	-----------

'DOWN' scrolls the entire screen down by one line, clearing the top line.

This statement is used to draw a "turtle graphics" figure.

<pre>DRAW 'flag' @ x,y USING array%(exp)          STATEMENT</pre>
<pre>'flag'  - designates the type of pixel action:            'S' signifies unconditional SET;            'R' signifies unconditional RESET;            'C' signifies pixel COMPLEMENT.</pre>
<pre>x,y     - is the coordinate of the starting point.            x is in the range (0-127 0-179); y is in            the range (0-47 0-71).</pre>
<pre>array%(exp) - is an integer array element.</pre>

Essentially, DRAW takes a list of line segment lengths combined with rotations, specified in any specified integer array at any point in the array (such as A%(10) or B%(18)), and plots a figure on the screen based on the list. The concept is very similar to turtle graphics in the LOGO language.

EnhComp DRAW allows 256 degrees of rotation and is properly scaled to assure minimal distortion of rotated figures. That is, a box will still look much like a box when it is rotated say 60/256s of a circle (60 DRAW degrees) and redrawn. Furthermore, the lengths of its sides will be close to that of the unrotated figure. In addition to allowing 256 degrees, DRAW allows noninteger line lengths and scaling: line lengths are specified in 1/256 graphics pixel width units.

To set up a turtle graphics figure, dimension an integer array to at least 4\*L-1, where 'L' is the required number of line segments needed to draw your figure. Each entry requires 4 bytes, encoded into a specified integer array (A in this example) in the following manner:

$A\%(x) = (\text{Byte}_1) + 256 * (\text{Byte}_2)$  where Byte\_1 is n/256 fraction of line length and Byte\_2 is the integer part of the line length. Bytes 1 and 2 contain the line length information: (BYTE 2) + (BYTE 1)/256 is the line length.

$A\%(x+1) = (\text{Byte}_3) + 256 * (\text{Byte}_4)$  where Byte\_3 specifies the rotation number in DRAW degrees (0-255) and Byte\_4 is the ENTRY code. Byte\_3 contains the number of degrees relative to the current orientation to draw the next line. The ENTRY code specified by Byte\_4 is determined from the following table:

Code Number	Signifying
0	List end; terminate DRAW
1	Draw line according to DRAW flag
2	Unconditionally SET line
3	Unconditionally RESET line
4	Unconditionally COMPL line
5 - 255	Ignore entry

Example Program:

```
10  DEFINT F
15  CLS
20  DIM FIGURE1(110)
25  Y=0
30  FOR X=0 TO 250 STEP 10
40  FIGURE1(Y)=X*6:'      Set line length = 6*X/256 units
50  FIGURE1(Y+1)=X+256:'  Rotation = X, entry code = 1
55  Y=Y+2
60  NEXT:'                Continue until figure completed
70  FIGURE1(Y+1)=0:'      Set 0 entry code to terminate list
75  '
77  ' Draw it!
79  '
80  DRAW SET@ 64,23 USING FIGURE1(0)
```

Notice that 'FIGURE1(0)' in line 80 above specifies the DRAW to begin interpreting entries at the first array entry. DRAW SET@ 64,23 USING FIGURE1(2) would skip drawing the first line in the figure specified by FIGURE1(0). Drawing begins at location (64,23) and the object is SET on the screen as per the DRAW flag 'SET'. DRAW RESET@ 64,23 USING FIGURE1(0) executed just after line 80 would immediately clear the figure off the screen.

This statement is used to terminate your program and return to DOS.

END	STATEMENT
-----	-----------

END causes a transfer back to DOS via the @EXIT address.

This function obtains the line number of the line containing an error.

ERL	- No operands are required!	FUNCTION
-----	-----------------------------	----------

'ERL' is usually used inside an error-trapping routine that was invoked by an error that occurred with an active 'ON ERROR GOTO'. If the line number is available, ERL returns the source line # in which the error happened.

This function obtains the error code of the last error generated.

ERR	- No operands are required!	FUNCTION
-----	-----------------------------	----------

'ERR' holds the code of the last error generated. As a consequence, it holds useful information only after an error occurs, which implies that an 'ON ERROR GOTO addr' must be active to override the standard error message and exit.

This statement is used for runtime program error control.

ERROR exp8	STATEMENT
exp8	- is a numeric expression which evaluates to the range (0-255).

The ERROR command forces a runtime error to occur. Normally, an error message 'RUNTIME ERROR CODE ccc IN SOURCE LINE #lllll' is printed and program execution is stopped. If an 'ON ERROR GOTO addr' is active, program execution branches to the address specified by the ON ERROR GOTO statement on occurrence of a runtime error. 'ON ERROR GOTO 0' disables this feature and causes the visual error message previously mentioned.

'EXISTS' will check for the availability of the designated filespec.

EXISTS(filespec\$)	FUNCTION
filespec\$ - specifies which file to look for.	

'EXISTS' will check if the specified file is available for use. It returns a logic TRUE (-1) if the file is accessible.

This function obtains the exponential of its argument.

EXP(exp)	FUNCTION
exp - is a numeric expression.	

EXP(exp) is equivalent to 2.7182818 ... raised to the 'exp'th power. If you're not familiar with this random-looking number, it pops up all over the place in engineering, scientific, and business problems. It returns, with full precision, a value of the same type given.

The 'FIELD' statement is used to assign the segments of a type "R" file record buffer to strings.

```
FIELD blknum,exp as var$<,exp2 as var2$> STATEMENT  
  
blknum - is file control block number, 1-15.  
  
exp - is the string length.  
  
var$ - is any string variable.
```

FIELD is used with "R" type files. It fields the record buffer into segments accessible by string variables, providing a means to read and write information in an orderly manner from or to any record in the file.

For writing to a file, information is placed into the FIELDed variables by means of the 'LSET' and 'RSET' commands. For obtaining non-string data read from fielded string variables, the 'CVI(var\$)', 'CVS(var\$)', and 'CVD(var\$)' functions are used.

Example Program:

```
5 CLEAR 1000  
10 ALLOCATE 1  
20 OPEN "R",1,"TEST/DAT"  
30 FIELD 1,256 AS A$  
40 LSET A$=STRING$(256,".")  
50 PUT 1,1  
60 CLOSE
```

Line 5 gives enough room for strings to breathe. Line 10 allocates a single file block. Line 20 opens the file for use; line 30 fields A\$ as entire record buffer (recall that EnhComp allows 32Kbyte length strings). Line 40 fills the record buffer with dots, and line 50 writes the record buffer to the first record in the file 'TEST/DAT', followed by the necessary CLOSE statement to neatly close the file and keep the disk directory running smoothly.

This function truncates the non-integer portion of its argument.

FIX(exp)	FUNCTION
exp - is a numeric expression.	

FIX returns the expression with the non-integer part stripped away. For example: FIX(-1.6) = -1.

These statements implement the typical FOR-NEXT loop construct.

FOR indexvar = start TO end <STEP step>	STATEMENT
NEXT <indexvar_1<,indexvar_2...>>	STATEMENT
indexvar - is a loop index variable.	
start	- is any numeric expression; the initial value of the loop index variable
end	- is any numeric expression; the terminating top or bottom limit value of the loop.
step	- is any numeric expression; added to the loop variable in each iteration. May be negative, in which case 'end' is bottom and not top limit.

'FOR' and 'NEXT' are the standard, eternal, BASIC looping construct statements. The 'FOR-NEXT' construct works by setting an index variable, specified in the initial 'FOR...' statement, to an initial value, unconditionally executing the loop code once (unless programming "tricks" are used) until a 'NEXT' is reached; then, unless the step was specified with 'STEP' in the 'FOR ...' setup, the step size is one, and this is added to the index variable. If the step is positive, 'NEXT' checks for 'indexvar' > 'toplimit'. If this is so, the statement following 'NEXT' is executed (the loop falls through). If 'indexvar' =< 'toplimit', 'NEXT' branches to the statement following the initial 'FOR...' setup, establishing a loop to be continued until 'indexvar' > 'toplimit'. Note that this might never happen, say if STEP = 0 and 'toplimit' > 'indexvar'.

If the step is negative, 'NEXT' checks for 'indexvar' < 'toplimit', the converse of the positive step case. Otherwise, the previous explanation holds true (exchanging '<' for '>' and vice versa.)

The desired loop variable(s) can be specified after a 'NEXT' statement. This is not necessary, however, except to preserve compatibility with interpretive BASIC programs. For instance, line 40 in the example program could simply be: 'NEXT:NEXT'.

Enhancement note: Double precision variables are allowed as loop indexes, something not allowed in interpretive basic.

For one example of the "programming trick" mentioned earlier, see "Programming idea #1" in the 'REPEAT-UNTIL' description.

Example Program:

```
5  CLS:PI = 3.14159
10 FOR R=1 TO 20 STEP 4:'   Radius of circle
20  FOR T = 0 TO 2*PI STEP PI/20:'   Parametric var.  in radians
30  X = R * 2 * COS(T)
40  Y = R * SIN(T)
50  SET(63+X,23-Y)
```

```
60 NEXT T:NEXT R: 'Could be: NEXT T,R
70 FOR X=127 TO 0 STEP -1
80 COMPL(X,23+SIN(X*8*PI/127)*15): 'Draw sine wave right to left
90 NEXT
```

This example program will draw a series of concentric circles on the screen.

This function obtains the amount of either the free stack space or the free string space.

FRE(exp)	FUNCTION
exp	- is either a STRING EXPRESSION, flagging FRE to return the amount of free string space left, or 0, flagging FRE to return MEM, the amount of free stack memory

The syntax box provides a complete explanation. 'FRE' is used, essentially, to determine the amount of space left for string storage. FRE(0) is numerically equivalent to MEM, described previously.

These statements are used to define multi-line user-defined functions.

```
FUNCTION name(input variable list)          STATEMENT
  statments
ENDFUNC                                     STATEMENT

input variable - is any simple string or numeric
                variable. Arrays are not allowed.

Note: FUNCTIONS are invoked via: "!name(args)"
```

DEFFN is used to define a function where a single "BASIC statement" can be entered on a single line to operate the function. It operates similarly to Interpretive BASIC. FUNCTION is a powerful statement that allows new multi-lined functions to be defined.

A user-defined multi-line function consists of three parts: A FUNCTION statement header; a user-function body; and the ENDFUNC statement. A defined function call is invoked by an exclamation point character followed by the function name and operand list, composed of any combination of numeric or string expressions separated by commas and enclosed in parentheses. For each operand there is a local variable in the function definition's input variable list. When a user-function call is made, the contents of the input variables are pushed onto the stack and then set equal to the specified operands.

Once the function computation is completed, the function value is returned with the statement 'RETURN value'. Any desired number of RETURNS can be included. A 'RETURN' statement without operands returns a value of 1.

As with user-commands, user-functions can be recursive, recursion depth limited only by free memory. Definitions may not be nested. Note that unlike Interpretive BASIC, user-functions are "defined" at compile-time and need not be executed to become "active"; in fact, definitions, if encountered, are skipped over.

Example Program #1:

```
10 INPUT"# TO TAKE FACTORIAL OF";X
20 PRINT X;"! = ";!FACTORIAL(X)
30 PRINT|GOTO 10
35 '
40 FUNCTION FACTORIAL(K)
50 IF K<2 THEN RETURN 1
60 RETURN K*!FACTORIAL(K-1)
70 ENDFUNC
```

The preceding program computes the factorial of a number using a recursive function. The recursive call takes place in line 60. The following program is simpler!

Example Program #2:

```
10 FOR X=1 TO 10
20 PRINT"X, SQUARE(X) | ";X,!SQUARE(X)
30 NEXT
```

```
40 END
45 '
50 FUNCTION SQUARE(K)
60 RETURN K*K
70 ENDFUNC
```

Consider the possibilities of directly using Z80 assembly language in a function definition. Here's one example|

Example Program #3:

```
10 INPUT"String to ENCODE";A$
20 B$=!ENCODE$(A$)
30 PRINT"Encoded string: ";B$: PRINT: GOTO 10
40 '
50 FUNCTION ENCODE$(T$)
60 '
70 ' Add 20 to each byte in string
80 '
90 Z80-MODE
100 LD IX,&(T$):' IX => String parameter block
110 LD C,(IX+0):LD B,(IX+1): LD L,(IX+2):LD H,(IX+3)
115 ' BC = string length, HL => String
120 "ENLOOP":LD A,B:OR C:JR Z,ENDENC:DEC BC
130 LD A,(HL):ADD A,20:' Number added is mostly arbitrary
140 LD (HL),A:INC HL:JP ENLOOP
150 "ENDENC"
160 HIGH-MODE
165 '
170 RETURN T$
180 ENDFUNC
```

The main point of the preceding program is the Z80 routine, not the simple encoding method (even a fairly dumb cryptographer could break this scheme in about five minutes). The speed of the efficient machine language routine makes the encoding time imperceptibly small for short strings. More complex, non-trivial encoding routines would benefit from the speed of a Z80 routine even more. Keep in mind that EnhComp allows strings of up to 32767 bytes in length.

If you copy the body of function ENCODE, modify ADD A,20 to SUB 20 and you have (guess what?) function DECODE (left as "an exercise for the reader").

'GET' reads a specified record into a record buffer.

GET blknum,recnum	STATEMENT
blknum	- is file control block number, 1-15.
recnum	- is the record number to read or write.

'GET' and 'PUT' are the two type "R" and type "X" disk file manipulation commands. PUT writes the contents of the record buffer to the specified record in the specified currently open file. GET reads a record from the specified currently open file into the record buffer.

Note that the 'recnum' operand is mandatory.

This statement allows your program to invoke unconditional program branching.

GOTO addr or GTO addr	STATEMENT
addr - is a line number or a label.	

GOTO is the standard BASIC way to transfer program execution to just about any desired point in the program. Either a conventional line number may be used, as with interpretive BASIC, or a label can be specified.

The following table describes the possible errors which could result from invalid use of this branch instruction:

Possible Errors	Reason
"UNDEFINED LINE"	Reference to undefined line #
"UNDEFINED LABEL"	Reference to undefined label

Example Program:

```
10 PRINT"This is the beginning ..."  
20 FOR X=0 TO 10:PRINT X,:NEXT:PRINT  
30 PRINT"AGAIN??"  
40 GOTO 10
```

In this program, the 'GOTO 10' in line 40 causes the example program to run on the computer indefinitely until someone comes along and BREAKs the program or the computer eventually crashes.

These commands allow your program to invoke unconditional program subroutine calls.

GOSUB addr or CSUB addr	STATEMENT
RETURN	STATEMENT
addr - is a line number or a label.	

GOSUB is the standard BASIC command to call a subroutine. Nested GOSUBs are limited only by available free stack memory. RETURN returns from a subroutine to the next instruction following the GOSUB invocation. Note the use of the POP command documented elsewhere. The following table describes the possible errors that could result from invalid use of these instructions:

Possible Errors	Reason
"UNDEFINED LINE"	Reference to undefined line #
"UNDEFINED LABEL"	Reference to undefined label

Line labels are a much better mnemonic device than line numbers, as well as being descriptive, as in the following example:

```
10 DIM A(10),B(10):' Note that ALL arrays must be dimensioned
20 FOR X=0 TO 10:A(X)=RND(X):B(X)=RND(0):?A(X),B(X):NEXT
30 GOSUB"SORT A":' Or: CSUB"SORT A"
40 GOSUB"PRINT A":' Could be GOSUB 140
50 GOSUB"SORT B"
60 GOSUB"PRINT B"
70 END
80 '
100 "SORT A":' Alternatively: JNAME"SORT A"
110 SCLEAR:KEY A(0):TAG B(0):SORT 11:RETURN
120 "SORT B"
130 SCLEAR:KEY B(0):TAG A(0):SORT 11:RETURN
140 "PRINT A"
150 FOR X=0 TO 11: PRINT A(X),B(X):NEXT:RETURN
160 "PRINT B"
170 FOR X=0 TO 11: PRINT B(X),A(X):NEXT:RETURN
```

This program loads arrays A() and B() with random numbers and then proceeds to sort them individually, first on A() with B() elements "tagging along", then on B() with A() as a TAG.

This function converts numeric expressions to strings of hexadecimal digits.

HEX\$(exp16)	FUNCTION
exp16	- is in the range <-32768 to 32767>

HEX\$ returns a 4 character ASCII hexadecimal representation of an integer.  
For example, HEX\$(-2) is equal to "FFFE".

These statements implement the typical IF-THEN-ELSE conditional structure.

IF cond THEN action <ELSE default action>	STATEMENT
IF cond program code	STATEMENT
<ELSE program code>	STATEMENT
ENDIF	STATEMENT

'IF-THEN-ELSE' comprise the critical conditional execution statements. EnhComp supports two forms of the 'IF-THEN-ELSE' construct: the standard single-line 'IF-THEN-ELSE' construct; and enhanced, multi-line 'IF-THEN-ELSE'. Here are two examples that are logically equivalent:

```
10  IF X<0 THEN A=A-X:K=1:IF A>16 THEN A=0 ELSE ELSE A=A+X
```

and

```
10  IF X<0
20    A=A-X:K=1
30    IF A>16
40      A=0
50    ENDIF
60  ELSE
70    A=A+X
80  ENDIF
90  PRINT"END OF CONDITIONAL CONSTRUCT"
100 END
```

The second example clearly shows the logical flow of the program, as opposed to the compact but visually linear first example. In the second example: If  $X < 0$ , line 20 ( $A=A-X$ ) is done. Line 40 ( $A=0$ ) is executed if the further conditional ( $A > 16$ ) at line 30 is met. Lines 60-80 are skipped are part of the ELSE code. If  $\text{NOT}(X < 0)$ , program flow goes to line 70 ( $A=A+X$ ) in the ELSE code section.

'INC' is used to increment an integer variable.

intvar - is either an integer variable or an integer array element.
--

'INC' and 'DEC' provide a very quick way to increment or decrement a specified integer variable, respectively.

Examples:

```
INC A%:      'A% = A% + 1
DEC B%(10):  'B%(10) = B%(10) - 1
```

This function will strobe the keyboard and returns the key depressed.

INKEY\$	FUNCTION
---------	----------

INKEY\$ returns a zero if no key is pressed or the key code if a key is pressed.

Example Program:

```
10 PRINT"Press any KEY to continue"  
20 A$=WINKEY$:IF A$="" THEN 20  
30 PRINT"Exiting program"  
40 END
```

This function obtains the value of the specified CPU port.

INP(portnum)	FUNCTION
portnum - specifies the CPU port in the range <0-255>	

INP performs a machine instruction to read the contents of the specified I/O port. It is the logical corollary to the 'OUT' command, described elsewhere, which sends a value TO to a specified CPU I/O port.

'INPUT' is used to accept keyboard input for variable value(s).

```
INPUT <@pos><"string";> var1 <,>var2 ...> STATEMENT  
var      - is any appropriate variable.
```

'INPUT' reads data from the keyboard. An optional "prompt" string may be printed. Leading blanks are skipped while reading. Strings (string variable specified) are read until a comma or a <CR> [CHR\$(13)] is reached. Numbers (numeric variable specified) are read until a space, a comma, or a <CR> is encountered.

'INPUT#' is used to read from a sequential file into variable(s).

INPUT#blknum, var1 <,var2 ...>	STATEMENT
blknum	- is a file control block number, <1-15>.
var	- is any appropriate variable.

'INPUT#' reads data from an "I" type file. Leading blanks are skipped while sequentially reading. Strings (string variable specified) are read until a comma, a <CR> [CHR\$(13)], or the end of the file is reached. Numbers (numeric variable specified) are read until a space, a comma, a <CR>, or the end of file is encountered.

This function will search a string for a designated substring.

INSTR(<exp,> exp1\$,exp2\$)	FUNCTION
exp1\$	- is the string to search.
exp2\$	- is the string to search for.
exp	- is an optional search start point

INSTR returns the position of a substring inside a string, if found; otherwise returning a 0. The beginning search point in the string can be optionally specified. If omitted, the search starts at the beginning of the string.

Example Program:

```
10 A$="THIS IS A TEST"
20 B$="IS"
30 I=1
40 F=INSTR(I,A$,B$)
50 IF F=0 THEN PRINT"END OF SEARCH.":END
60 PRINT B$;" FOUND IN ";A$;" AT POSITION ";F
70 I=F+1:GOTO 40:' Continue search
```

This is the "greatest integer" function.

INT(exp)	FUNCTION
exp - is a numeric expression.	

INT works with any precision expression, returning the same precision. It returns the greatest integer less than 'exp'. For the confused, some examples:

```
INT(3.4) = 3
INT(.5) = 0
INT(-.5) = -1
INT(-1.4) = -2
```

This statement is used to invert all graphics on the video screen.

INVERT	STATEMENT
--------	-----------

This inverts all graphics on the screen. SET points are RESET and vice versa.  
Text (characters not within range 128 =< x =< 191) is ignored.

This statement is used to establish a line label.

JNAME"label" or "label"	STATEMENT
label	- is a (unique) string of characters representing a memory location.

Labels are used to establish branch points for use with GOTOs, GOSUBS, or any BASIC statement. The following table describes the possible error which could result from invalid use of these statement:

Possible Error	Reason
"MULTIPLY DEFINED SYMBOL"	Two or more labels defined (via JNAME"label" or "label") are equivalent

Example Program:

```
10 DIM A(10),B(10):' Note that ALL arrays must be dimensioned
20 FOR X=0 TO 10:A(X)=RND(X):B(X)=RND(0):?A(X),B(X):NEXT
30 GOSUB"SORT A":' Or: CSUB"SORT A"
40 GOSUB"PRINT A":' Could be GOSUB 140
50 GOSUB"SORT B"
60 GOSUB"PRINT B"
70 END
80 '
100 "SORT A":' Alternatively: JNAME"SORT A"
110 SCLEAR:KEY A(0):TAG B(0):SORT 11:RETURN
120 "SORT B"
130 SCLEAR:KEY B(0):TAG A(0):SORT 11:RETURN
140 "PRINT A"
150 FOR X=0 TO 11: PRINT A(X),B(X):NEXT:RETURN
160 "PRINT B"
170 FOR X=0 TO 11: PRINT B(X),A(X):NEXT:RETURN
```

This program loads arrays A() and B() with random numbers and then proceeds to sort them individually, first on A() with B() elements "tagging along", then on B() with A() as a TAG.

'KILL' will delete the designated file from the disk directory.

KILL"filespec\$"	STATEMENT
filespec\$ - designates the file to remove.	

'KILL' removes the disk directory entry of a file and frees up the space that the data of the file took on the disk.

This statement is used to scroll the video screen left one column.

LEFT	STATEMENT
------	-----------

This statement scrolls the entire screen left by one character. The entire last screen column is cleared, and the entire 0th column is written over with the first column.

This function parses a substring of a string.

LEFT\$(exp\$,exp1)	FUNCTION
exp\$	- is any string expression.
exp1	- is the number of leftmost characters to use for the obtained substring.

LEFT\$ chops a substring from the left of a string. For example:

```
LEFT$("FOUR SCORES",4) = "FOUR"  
LEFT$("NO MUSAK",6)   = "NO MUS"
```

Note that MID\$ can easily simulate LEFT\$. For example, LEFT\$(exp\$,exp) is equivalent to MID\$(exp\$,1,exp) assuming len(exp\$) >= exp.

This function obtains the length of its string argument.

LEN(exp\$)	FUNCTION
exp\$ - is any string expression.	

LEN returns the length of the specified string expression. Naturally, the string expression can be a single string variable. For example,

```
A$ = "TEST"  
A = LEN(A$)
```

assigns 4 to 'A'. And:

```
A$ = "TEST"  
A = LEN(A$ + "ING")
```

assigns 7 to 'A'. (A quicker way would be: A=LEN(A\$)+3.)

'LET' is used to assign a value to a variable.

<LET> var = exp	STATEMENT
var	- is any variable.
exp	- is any expression of appropriate type.

Any variable assignment can be done without the LET command. 'LET' is included to preserve compatibility.

Examples:

```
A = 10:'      Assign 10 to variable A
A$ = "HELLO":' Set A$ to "HELLO"
```

Note on "Garbage collection" and string variables: Interpretive BASIC on the TRS-80 is notorious for the string "garbage collection" lock-up that occurs when free string space is needed and it is necessary to clean up the garbage left over from previous string manipulations. EnhComp compiled programs don't suffer from this malady. There is never "garbage" lying around in the string storage area; the only time extensive re-arrangement of strings and string pointers can occur is during a string assignment.

'LINEINPUT' is used to accept keyboard input into a string.

```
LINEINPUT <@pos><"string";>var1$<,var2$...> STATEMENT  
var$      - is any appropriate string variable.
```

'LINEINPUT' reads data from the keyboard without the usual "?" prompt. An optional "prompt" string may be printed. Leading blanks are skipped while reading. The input line is read verbatim until a <CR> [CHR\$(13)] is reached.

'LINEINPUT#' is used to read from a sequential file into a string.

LINEINPUT#blknum, var\$ <,var2\$ ...>	STATEMENT
blknum	- is a file control block number, <1-15>.
var\$	- is any string variable.

LINEINPUT# reads a string from an "I" type file. All characters starting at the current read point up to a <CR> [CHR\$(13)] or the end of file are read into the string, up to the limit of 255 characters.

This statement is used to set the number of printed lines per page.

```
LINEPAGE = exp                                STATEMENT
exp      - is a numeric expression which evaluates to
           the range <2-255>.
```

This statement sets the number of lines printed on a page until automatic Top Of Form (TOF) occurs.

This statement is used to set the printer's left hand margin.

LMARGIN = exp	STATEMENT
exp	- is a numeric expression which evaluates to the range <2-255>.

This statement sets the number of spaces automatically printed when a carriage return (ASCII 13) is sent to your printer. The default is 0 spaces.

This statement will load a '/CMD' type program from disk into memory.

LOAD"filespec\$"	STATEMENT
filespec\$ - designates the file to load.	

'LOAD' loads a machine language program from disk into memory without executing it.

This function obtains the natural logarithm of its argument.

LOG(exp)	FUNCTION
exp	- is a numeric expression.

LOG returns the natural logarithm of an expression. Theoretically (ignoring inevitable round-off error),  $\text{LOG}(\text{EXP}(\text{exp})) = \text{exp}$ . 'LOG' returns, with full precision, a value of the same type given (ex.:  $\text{LOG}(1.7\#\text{X}\#)$  returns the log of this expression accurate to 16 decimal digits due to the double precision.)

This statement is used to print data to the printer.

LPRINT <item> <', '>:<', '>:<'TAB(exp)'\> ... STATEMENT	
item	- is a "stringliteral" or a numeric / string expression
<, ;>	- are delimiters

All LPRINT statements used in TRS-80 interpretive BASIC programs should compile and function with equivalence with no modifications necessary.

Note that PRINT output can be sent to either the printer, the screen, or a disk file using the 'PRINT#' statement depending on the value of the expression chosen in the statement: 'PRINT#exp,...'. For example, the same section of code could be used for both screen and printer output simply by changing the value of a variable and calling the same subroutine:

```
.  
. .  
90 "BPRINT"  
100 F=0:GOSUB "PRINT":' Send to screen  
110 F=-3:' Send to printer  
120 "PRINT"  
130 PRINT#F,"TO: ";FRIEND$  
140 PRINT#F,"FROM: ";SENDER$  
150 RETURN
```

Default SCREEN or PRINTER TAB positions can be altered with the SZONE and PZONE commands respectively documented elsewhere in this manual. A comma delimiter or equivalently TAB(255) tabs the cursor to the next screen or printer zone, depending on the current output mode.

USING is now a string expression. Compiled and interpreted BASIC 'PRINT USING' statements usually produce the same output.

'LSET' is used to set information into FIELDed string variables for use with random access files.

LSET var\$ = exp\$	STATEMENT
var\$	- is FIELDed string to which the information is to be added.
exp\$	- is the information to add.

'LSET' and 'RSET' are really just versions of 'MID\$ ='. Their main intended purpose is to set information into FIELDed string variables. FIELDed strings must point to a static memory location (in a file's record buffer).

For 'LSET', var\$ is overlaid starting at position 0 with exp\$, filling any remaining portion of var\$ with blanks (ASCII 32). For 'RSET', var\$ is overlaid with exp\$, measuring from the end of var\$, filling any remaining portion of var\$ with blanks (i.e. the information is "right justified").

A standard string assignment, such as A\$="MONDAY" places A\$'s data in the string storage area, which is constantly changing. LSET and RSET (and MID\$) directly alter existing a string variable's contents without changing the string's position in memory. The main difference between MID\$ and LSET/RSET is that the latter commands fill the remaining characters in the affected string with blanks, or CHR\$(32)'s.

Note that compiled LSET and RSET, as with interpretive Disk BASIC LSET/RSET commands, work on any string variable, not just FIELDed string variables.

Examples (in all examples A\$ is 10 chars long):

LSET A\$="HELLO":'	Now A\$="HELLO      "
LSET A\$="12345678912":'	Now A\$="1234567891"
RSET A\$="HELLO":'	A\$="      HELLO"
LSET A\$=MKD\$(1.2345#):'	Now first 8 bytes of A\$ contain the floating point double precision number 1.2345#

This function obtains the amount of free stack space.

MEM	- No operand is required!	FUNCTION
-----	---------------------------	----------

'MEM' simply returns the amount of free memory left for array dimensions, ALLOCATE, etc. -- or what amounts to the same thing, the free stack space left.

The 'MID\$' statement is used to overlay a string or portion of a string with another string.

MID\$(var\$,exp1 <,exp2>) = exp\$	STATEMENT
var\$	- is string to be modified.
exp1	- is the starting position of var\$ to be overlaid by exp\$.
exp2	- designates how many characters of exp\$ will overlay the string, var\$.
exp\$	- is the overlaying string.

MID\$ is the only reserved word used as both a function and a command. Don't confuse the MID\$ function with MID\$ statement, although they perform similar operations. MID\$ operates directly on string variables. MID\$ never changes the length of the string variable.

Examples:

```
A$="ABCDE": MID$(A$,1)="xyz":'      Now A$ = "xyzDE"
A$="ABCDE": MID$(A$,2,2)="xyz":'    Now A$ = "AxyDE"
A$="ABCDE": MID$(A$,1,4)="xyz":'    Now A$ = "xyzDE"
A$="ABCDE": MID$(A$,1)="1234567":'  A$ now = "12345"
```

Example 1 is straightforward. In example 2, the optional length expression of two limits the number of characters overlaid from the expression "xyz". In example 3, although the maximum length was specified as 4, the length of "xyz" is only 3. In example 4, A\$ is too short to contain the entire string expression.

This function parses substrings of a string.

MID\$(exp\$,exp1 <,exp2>)	FUNCTION
exp\$	- is any string expression.
exp1	- is the starting position.
exp2	- is the optional substring length. If exp2 is omitted, the rest of exp\$ after exp1 is taken

Virtually all BASIC's have a string function performing equivalently to the MID\$ function. MID\$ can pull any desired substring from a given string. For example:

```
MID$("ABCDEF",2,3) = "BCD"  
MID$("BYEBYE",4,2) = "BY"  
MID$("HOUSE",2)   = "OUSE"
```

Note that MID\$ can easily simulate both LEFT\$ and RIGHT\$. For example:

```
LEFT$(exp$,exp) is equivalent to MID$(exp$,1,exp)  
RIGHT$(exp$,exp) is equivalent to MID$(exp$,len(exp$)-exp+1)
```

assuming len(exp\$) >= exp.

These functions convert numeric expressions to their packed string representation.

MKD\$(exp)	FUNCTION
MKI\$(exp)	FUNCTION
MKS\$(exp)	FUNCTION
exp	- is a numeric expression of the desired type

MKD\$ maps a double precision number to an 8-byte string. The primary purpose of MKD\$ is to store double precision numbers in random access disk files, since FIELD statements accept strings only. Similarly, MKI\$ maps an integer to a 2-byte string for storing integers and MKS\$ maps single precision numbers to 4-byte string for storing single precision expressions.

For example:

```
10 A#=1.2345678#: B=2.71828
20 ALLOCATE 1:OPEN "R",1,"TEST/DAT"
30 FIELD 1,8 AS PY$,4 AS E$
40 LSET PY$=MKD$(A#): LSET E$=MKS$(B)
:
```

The string-encoded contents of A# are LSET into the first 8 bytes of the record buffer, effectively storing A#, and 'B' is stored in the next 4 bytes after that. The program could go on to make other LSETs and RSETs, then write the buffer to a record and close the file.

This statement allows your program to invoke conditional branching and sub-routine calls.

```
ON exp GOTO addrlist          STATEMENT
ON exp GOSUB addrlist        STATEMENT
exp      - designates the branch index of 'addrlist'.
addrlist - is a list of line numbers or labels
```

'ON ... GOTO' substitutes for a long list of compares and GOTOs. The 'exp' indexes the line number or label address list. If there are fewer than 'exp' addresses in the list, the statement following the 'ON ... GOTO/GOSUB' is executed.

Example Program:

```
5   REM
10  REM  Simplified counting schema ...
20  REM
30  REM  (Note: unsuitable for check-writing routines)
40  REM
45  FOR X=1 TO 5
50  ON X GOTO 100,200,300
55  PRINT"MANY"
60  NEXT
70  PRINT"...":END
100 PRINT"ONE":GOTO 60
200 PRINT"TWO":GOTO 60
300 PRINT"THREE":GOTO 60
```

This statement is used to provide <BREAK> key control of your program.

ON BREAK GOTO addr	STATEMENT
addr - is either a LINE # or a LABEL	

'ON BREAK GOTO addr' causes a jump to the specified line number or label if the <BREAK> key is hit and the BREAK scan is activated. 'ON BREAK GOTO 0' disables <BREAK> key branching, parallel to 'ON ERROR GOTO 0'. Causing an 'ON BREAK GOTO addr' jump also automatically disables <BREAK> key branching.

'BKON' and 'BKOFF' can be used to effectively turn the BREAK key on or off, respectively. They affect only the BREAK scan flag. BKON will have no apparent effect if the "-NX" directive flag has been specified, since the BREAK scan code calls will be left out of the compiled program.

#### Example Program

```
5   ON BREAK GOTO 100
10  PRINT"HO HUM ..."
20  FOR X=0 TO 1E12: NEXT
30  PRINT"OH BOY, LET'S COUNT TO A QUADRILLION NOW!"
40  END
100 PRINT"THANKS! SAVED FROM A FATE WORSE THAN SCARFMAN...."
```

This statement is used for runtime program error control.

<pre>ON ERROR GOTO addr</pre>	<pre>STATEMENT</pre>
<pre>addr</pre>	<pre>- is either a line number or a label which specifies the target of the branch.</pre>

Normally, an error message,

```
RUNTIME ERROR CODE ccc IN SOURCE LINE #lllll
```

is printed and program execution is stopped when a runtime error is detected. If an 'ON ERROR GOTO addr' is active, program execution branches to the address specified by the ON ERROR GOTO statement on occurrence of a runtime error. 'ON ERROR GOTO 0' disables this feature and causes the visual error message previously mentioned.

The ERROR command can be used to force a runtime error to occur (usually used to certify the correctness of your error trapping routine).

'OPEN' is used to prepare a disk file for input or output operations.

OPEN "type\$",blknum,"filespec\$"<,reclen>	STATEMENT
type\$	- specifies the type of file access desired: R,r for RANDOM ACCESS filetype; O,o for SEQUENTIAL OUTPUT filetype; E,e for SEQUENTIAL OUTPUT EXTENDED; I,i for SEQUENTIAL INPUT filetype; X,x for EXTENDED RANDOM ACCESS filetype.
blknum	- is the file control block to use, in the range <1-15> (see ALLOCATE).
filespec\$	- is the name of the disk file to access.
reclen	- is an optional expression in the range (1-255) designating the number of bytes in each record of the file to be opened. This Must match the previous record length if the file already exists.

Before a disk file can be manipulated it must first be OPENed. Also, before any file can be opened, space for the total number of simultaneously open files must be allocated using the 'ALLOCATE' statement. This is similar to the function of specifying the maximum number of files via the "F=files" parameter used when invoking a BASIC interpreter.

There are five allowable file types; however, there are really only three fundamental types of files: Random Record Access, Sequential, and list-directed Extended. The file type string character may be upper or lower case.

Random access, specified by an "R" type in the OPEN statement, implies that file manipulation will be done discretely with any selected individual record in the file via the GET (get/read record) and PUT (put/write record) commands, which are described in detail elsewhere in this manual.

With sequential access, a file is read ("I") or written ("O" or "E") sequentially, basically a byte at a time, with INPUT# or PRINT#, respectively. EnhComp prepares a type "E" file by positioning it to its end as soon as it is opened. This permits you to extend the file by appending new information to the existing data. Type "E" can also be specified for a "new" file.

The 'POSFIL' command described elsewhere can set the read or write (determined automatically by file type) position to any point in a sequential file (limited by existing file size in "I" mode, free disk space in "O" mode).

EnhComp supports a fairly powerful new random access file mode, "X". This extended mode allows the use of lists of simple variables as field specifiers rather than the cumbersome, difficult to conceptualize conventional FIELD statement.

Extended file mode uses the usual 256 byte LRL disk random record length but allows logical record lengths of from 1 to 32767 bytes long. This record length is defined at open time, with the statement:

```
OPEN "X",blknum,"filename$",reclen
```

where 'reclen' is the desired record length. Note that this record length is entirely the responsibility of the programmer to track; it is entirely possible to close a previously opened and written-to extended file and open it again with a different record length. No explicit error will occur.

The record structure is defined with the XFIELD statement, rather than the FIELD statement as is the case for "R" file types. Its format allows either numeric or string variables in its list. Array variables are not allowed in the list.

This command is used to send a value to a specified CPU port.

OUT portnum,value	STATEMENT
portnum	- is a numeric expression which evaluates to the range <0 to 255>, specifying a CPU port number.
value	- is a numeric expression which evaluates to the range <0 to 255>, specifying a byte to be sent out the port.

OUT provides a means to send information to any of the CPU I/O ports. The assembler can also accomplish this as a matter of course by assembling a native code OUT instruction directly.

This statement is used to set the physical printer page length.

PAGELEN = exp	STATEMENT
exp	- is a numeric expression which evaluates to the range <2-255>.

This statement sets the printer page length for use with all printing operations. Note this is the physical length, in lines, of your printed page.

This statement is used to fill in a bounded shape.

	STATEMENT
<code>PAINt(x,y)&lt;,color&gt;</code>	
<code>x,y</code>	- is the coordinate of a point interior to the bounded shape.
<code>color</code>	- is the color used to fill the shape <0,1> where black = 0 and white = 1. If color is omitted, it will default to 1.

PAINt can be used to fill in any shape defined by a boundary of pixels of the same color as the "color" operand. The point "x,y", entered in pixel coordinate values, must be interior to the bounded shape.

The following example will plot a triangle then fill in the interior of the triangle.

```
05 CLS
10 PLOT S,10,10 TO 120,10
20 PLOT S,50,40 TO 120,10
30 PLOT S,10,10 TO 50,40
40 PA=1:PAINT(55,15),PA
50 A$=WINKEY$
```

This function obtains the byte stored at a memory address.

PEEK(exp16)	FUNCTION
exp16	- represents a memory address in the range <-32768 to 32767>.

'PEEK' is a means to "peek" directly into any selected byte in the computer's memory. For example, on the TRS-80 Model I/III, PRINT PEEK(0) prints a 243 (from ROM), or the Z80 instruction "DI", disable interrupts, the first instruction executed on power up.

This statement is used to plot a line of pixels.

PLOT 'flag',x1,y1 TO x2,y2	STATEMENT
x1,y1	- specifies the coordinate point of one line endpoint.
x2,y2	- specifies the coordinate of the other line endpoint.
'flag'	- designates the type of pixel action: 'S' signifies unconditional SET; 'R' signifies unconditional RESET; 'C' signifies pixel COMPLEMENT.

PLOT is a statement that allows an entire line to be drawn at once. It can SET, RESET, or COMPL a line on the screen. For example:

```
PLOT S,0,0 TO 127,47
```

would set a line between (0,0) and (127,47).

```
PLOT R,127,47 TO 0,0
```

would reset that same line. And,

```
PLOT C,127,0 TO 0,47
```

plotted after 'PLOT S,0,0 TO 127,47' was executed would produce a line going from the upper right hand corner of the screen to the lower left, resetting the dots where it intersected in middle of the line drawn from the upper left corner to the lower right. The following program makes an interesting fanline pattern on the screen.

```
10 FOR Y=0 TO 47 STEP 3
20 PLOT S,0,0 TO 127,Y: 'Draw line from (0,0) to right edge
30 PLOT S,127,47 TO 0,47-Y: 'Draw line from (127,47) to left edge
40 NEXT
```

This function obtains the point value of the specified pixel location.

POINT(x,y)	FUNCTION
x,y	- is the coordinate of the pixel. 'x' is in the range <0-127 or 0-179> and 'y' is in the range <0-47 or 0-71>

'POINT' checks whether any selected graphics pixel on the screen is set or not. It returns -1 if the point is SET, 0 otherwise.

This statement is used to poke a value into a memory location.

POKE exp16,exp8	STATEMENT
exp16	- specifies a memory address in the range <-32768 to 32767>.
exp8	- is a numeric expression which evaluates to the range <0 to 255>.

POKE and WPOKE allow direct modification of any RAM location in memory. WPOKE "pokes" two bytes at a time in conventional low order/high order format into the specified address, whereas POKE inserts only a single byte.

This statement is used to escape from a GOSUBed subroutine.

POP	STATEMENT
-----	-----------

POP is a quick and dirty way to get out of a messy situation whilst stuck in the middle of a subroutine. It erases all effects of the last GOSUB from the stack, allowing clean error recovery, or whatever. This 'POP' operation is not to be confused with the CPU opcode, POP.

Example Program:

```
10 GOSUB 20:PRINT"RETURNED AND BACK TO 10":END
20 GOSUB 30:PRINT"LINE 20. RETURNING TO 10.":RETURN
30 PRINT"LINE 30. 'POP' and 'RETURN'."
40 POP:RETURN
```

The POP at line 40 wipes out the GOSUB at line 20, causing the RETURN directly following the POP to return to the next-lesser-level of GOSUB, the one made in line 10.

This function returns the current position of the cursor relative to the start of the line it appears on.

POS(dummy exp)	FUNCTION
----------------	----------

'POS' returns the current column position of the cursor. For instance:

```
PRINT:PRINT"HELLO";:A=POS(0)
```

assigns 'A' to 5, the cursor position after 'HELLO' is printed.

'POSFIL' allows you to position a sequential input/output file pointer for subsequent I/O operations.

	STATEMENT
POSFIL(#blknum,recnum,offset)	
blknum	- is a file control block number, <1-15>.
recnum	- is the disk file's 256-byte record number.
offset	- is the offset within the record, <0-255>

'POSFIL' is useful for positioning the sequential input/output pointer for selective sequential reading and writing. As with 'RDGOTO', which selects any 'DATA' statement in a program for the next 'READ', 'POSFIL' is the equivalent extension for sequential files.

Example Programs:

```
10 ALLOCATE 1:OPEN "O",1,"TEST/DAT"  
20 POSFIL(#1,2,0):PRINT#1,"HELLO":CLOSE
```

The string 'HELLO' is written from the beginning of the second record in the file, as opposed to the default of the start of the first record.

```
10 ALLOCATE 1:OPEN "I",1,"TEST/DAT"  
20 POSFIL(#1,5,67):INPUT#1,A$:CLOSE
```

A\$ is sequentially read, starting from the 67th character of the fifth record in the file 'TEST/DAT' (assuming that TEST/DAT contains at least 5 records).

This statement is used to print data to the video screen.

```
PRINT <'#' numexp,>:<'@' screenpos,> <item> STATEMENT
      <','>:<'>:<'>:<'TAB(exp)'\> ...

numexp   - is a numeric expression within the range
           <-3 to 15>: -3, send to PRINTER; 0, send to
           VIDEO display; 1 thru 15, send to disk file

screenpos - is a numeric expression between 0 and 1023
           specifying a new cursor relative position

item      - is a "stringliteral" or a numeric / string
           expression

< , ; >   - are delimiters
```

All PRINT statements used in TRS-80 interpretive BASIC programs should compile and function with equivalence with no modifications necessary.

PRINT output can be sent to either the printer, the screen, or a disk file depending on the value of the expression chosen in the statement: 'PRINT#exp,...'. For example, the same section of code could be used for both screen and printer output simply by changing the value of a variable and calling the same subroutine:

```
.
.
.
90 "BPRINT"
100 F=0:GOSUB "PRINT":' Send to screen
110 F=-3:' Send to printer
120 "PRINT"
130 PRINT#F,"TO: ";FRIENDS$
140 PRINT#F,"FROM: ";SENDER$
150 RETURN
```

Default SCREEN or PRINTER TAB positions can be altered with the SZONE and PZONE commands respectively documented elsewhere in this manual. A comma delimiter or equivalently TAB(255) tabs the cursor to the next screen or printer zone, depending on the current output mode.

USING is now a string expression. Compiled and interpreted BASIC 'PRINT USING' statements usually produce the same output.

'PRINT#' is used to write to a sequential file.

PRINT#blknum, ...	STATEMENT
blknum - is a file control block number, <1-15>.	

'PRINT#' writes data to an "O" or "E" type file. Except for 'PRINT@', information following the 'PRINT#blknum', and output from it, is in the same format as a screen 'PRINT' statement, except that output is routed to a file instead of to the screen.

Note that EnhComp allows you to direct the output of a 'PRINT#blknum' command to be directed to either the video screen or your printer by specifying 'blknum' as 0 or -3 respectively. Thus, the command:

```
'PRINT#-3,"This is a test"'
```

will print the text string on your printer. Expressing the 'blknum' as a variable permits you to designate the output device at runtime.

'PUT' writes a record buffer to a specified record.

PUT blknum,recnum	STATEMENT
blknum	- is file control block number, 1-15.
recnum	- is the record number to write.

'GET' and 'PUT' are the two type "R" and type "X" disk file manipulation commands. PUT writes the contents of the record buffer to the specified record in the specified currently open file. GET reads a record from the specified currently open file into the record buffer.

Note that the 'recnum' operand is mandatory.

This statement is used to set the line printer print zones.

PZONE(pos 1,...,pos n)	STATEMENT
PZONE(*)	
pos	- is a numeric expression between 0 and 255 which designates printer tab positions.

PZONE sets up default printer TAB positions for LPRINT (or PRINT#-3) ",  
modifiers. PZONE(\*) clears all printer stops.

This statement seeds the random number generator.

RANDOM <exp>	STATEMENT
--------------	-----------

RANDOM reseeds the "random" number generator to assure a high probability of a non-repeating "random" sequence of numbers. EnhComp uses the well known and often used method of linear congruential modulus to generate random numbers. To assure high randomness and high non-repeatability, double precision variables are used. This accounts for the relatively slow speed of the RND function. However, randomness is tremendously improved over TRS-80 BASIC RND results (as well as many other languages with poor random number generators.)

The seed which is used will be a random number between <0-255> if no operand is given; else it is seeded with the given operand. Specifying a particular seed value will start the same sequence every time for any given operand, which can be between 0 and about 2,400,000.

This statement allows you to reset the DATA list pointer.

RDGOTO addr or RDGTO addr	STATEMENT
addr - is either a line number or a label.	

DATA provides an efficient way to store many static pieces of data in a program (such as a tax table). Executing a DATA statement does nothing as program execution jumps over the data list. The data list is read into variables with the READ statement. READ normally reads data starting from the beginning of the list.

RESTORE and RDGOTO provide ways to point at the desired data list. RDGOTO, especially, eliminates the wasteful process of reading and discarding lists of data to get to the desired list required in interpretive BASIC.

Initially, the first data item read, unless the data pointer is changed by a RDGOTO/RDGTO statement, will be the first data item in the first DATA statement in the program.

Example Program:

```
5   RDGOTO "PRIME"
10  READ TITLE$:PRINT TITLE$:PRINT:READ N
20  FOR X=1 TO N:READ A:?A,:NEXT
30  END
35  '
40  "FIB"
50  DATA The first EIGHT Fibonacci numbers in order
60  DATA 8, 1,1,2,3,5,8,13,21
70  "PRIME"
80  DATA The first NINE prime numbers in sequential order
90  DATA 9, 2,3,5,7,11,13,17,19,23
```





These statements implement the typical REPEAT-UNTIL loop construct.

REPEAT	STATEMENT
UNTIL exp	STATEMENT
exp	- is any numeric expression (usually boolean)

'REPEAT-UNTIL' is a looping construct found in some "structured" languages such as PASCAL. As with 'FOR-NEXT', unless unusual programming techniques are used, the loop is unconditionally executed one time. Consider the fact that unlike many compilers EnhComp allows more than one 'UNTIL' or 'NEXT' for a single 'REPEAT' or 'FOR' statement, respectively. Runtime program flow might (often does) variably choose a particular 'UNTIL' or 'NEXT' to branch to, rendering compile-time selection impossible.

The 'REPEAT' statement flags a point to loop to when the next 'UNTIL' is encountered and its expression is non-zero. That is, a loop is made when the expression following the 'UNTIL' is boolean TRUE (non-zero on the TRS-80). Program execution resumes at the statement following 'UNTIL exp' if 'exp' = 0 (the loop falls through.)

Example Program:

```
10 INPUT"LETTER (A-Z) TO STOP FOR";S$
20 REPEAT
30 T$=CHR$(RND(26)+64)
40 PRINT T$,
50 UNTIL S$=T$
```

This prints a random letter until the user-selected letter is encountered.

Programming Idea #1

There is a trick that may be used to defer execution of a loop even a single time, with either 'FOR-NEXT' or 'REPEAT-UNTIL'. The trick involves the use of the user-defined command mechanism, and goes as such:

First a look at FOR-NEXT. The required input variables are: 1) The initial loop index variable value, 2) the top limit, and 3) the step size. Clearly, some of these may be deferred if desired by setting some of them to constants. Then, define a user-command like so:

```
10 %LOOP0(0,10,.25): 'Will perform FOR TEST=0 TO 10 STEP .25
20 %LOOP0(10,0,1): 'Nothing will happen because 10 > 0
30 END
50 '
100 COMMAND LOOP0(IVALUE,TOPLIM,INCR)
150 '
200 IF INCR<0
300 IF IVALUE>TOPLIM THEN RETURN
400 ELSE
500 IF IVALUE<TOPLIM THEN RETURN
600 ENDIF
650 '
```

```
700 FOR TEST = IVALUE TO TOPLIM STEP INCR
... ..
... NEXT

... RETURN
... ENDCOM
```

(Naturally, the line numbering is arbitrary -- they could be any other sequential allowable numbers). 200-600 prevents the loop from being started at all if the initial index variable value falls outside of the specified limit.

Without a doubt you can see how to apply this idea to 'REPEAT-UNTIL' loops. One idea: set up the user-command to accept a list of critical variables used in the 'UNTIL' expression. Then, apply the pre-loop-check to the 'UNTIL' expression. If zero, then RETURN, otherwise, march onwards. For example:

```
COMMAND LOOP1(A,B,C)
D = 64
IF (A+B) > (C+D) THEN RETURN
REPEAT
PRINT A
A = A + B
UNTIL A > (C+D)
RETURN
ENDCOM
```

This statement is used to turn off a pixel.

RESET(x,y)	STATEMENT
x	- is a numeric expression which evaluates to the range <0 - 127> for 64-column screens and <0 - 159> for 80-column screens.
y	- is a numeric expression which evaluates to the range <0 - 47> for 16-row screens and <0 - 71> for 24-row screens.

SET, RESET, and COMPL form the set of the single-pixel-affecting graphics commands. Note that screens that display 16 rows of 64 characters will display 72 rows by 160 columns of graphics pixels; screens that display 24 rows of 80 characters will display 72 rows by 160 columns of graphics pixels.

SET is a standard TRS-80 BASIC command that unconditionally turns on any selected graphics pixel on the TRS-80 screen. The RESET command turns a pixel OFF. The COMPL command complements a selected graphics pixel, turning it ON if it is OFF and vice versa. A function, POINT(x,y), which is related to the pixel graphics commands is discussed in the section on functions.

The following illustrates a brief example of these graphics commands:

```
5   Y=23:RANDOM:CLS
10  FOR X=0 TO 127
20  SET(X,Y)
30  Y=Y+SGN(RND(3)-2)
40  IF Y<0 THEN Y=0 ELSE IF Y>47 THEN Y=47
50  NEXT
60  FOR X=0 TO 127
70  COMPL(X,23):NEXT
80  FOR X=0 TO 127
90  RESET(X,23):NEXT
```

The program first plots a pseudo-"mountainous" profile on the screen, proceeds to "complement" all graphics dots down the middle of the screen, and finally resets all pixels through the middle of the screen.

This statement allows you to reset the pointer of a data list.

RESTORE	STATEMENT
---------	-----------

DATA provides an efficient way to store many static pieces of data in a program (such as a tax table). Executing a DATA statement does nothing as program execution jumps over the data list. Initially, the first data item read will be the first data item in the first DATA statement in the program.

After some data items in the list have been read, the RESTORE statement may be used to reset the list pointer to the beginning of the table. RDGOTO can be used to reposition the list pointer to any labeled location of the data list. This eliminates the wasteful process of reading and discarding lists of data to get to the desired list required in interpretive BASIC.

Example Program:

```
5   RDGOTO "PRIME"  
10  READ TITLE$:PRINT TITLE$:PRINT:READ N  
20  FOR X=1 TO N:READ A:?A,:NEXT  
30  END  
35  '  
40  "FIB"  
50  DATA The first EIGHT Fibonacci numbers in order  
60  DATA 8, 1,1,2,3,5,8,13,21  
70  "PRIME"  
80  DATA The first NINE prime numbers in sequential order  
90  DATA 9, 2,3,5,7,11,13,17,19, 23
```

This statement performs an unconditional program branch.

RESUME addr	STATEMENT
addr - is a line number or a label.	

'RESUME addr' is precisely equivalent to 'GOTO addr' (see description elsewhere in this manual). It is implemented to preserve some compatibility with interpretive BASIC programs using this command.

**INCOMPATIBILITY NOTE**

**The RESUME NEXT interpretive BASIC feature is not supported by EnhComp Ver. 2.x.**

This statement is used to return from a GOSUBed subroutine.

RETURN	STATEMENT
--------	-----------

GOSUB is the standard BASIC command to call a subroutine. Nested GOSUBs calls are limited only by available free stack memory.

RETURN returns from a subroutine to the next instruction following the GOSUB invocation. Note the use of the POP command documented elsewhere.

Example Program:

```
10 DIM A(10),B(10):' Note that ALL arrays must be dimensioned
20 FOR X=0 TO 10:A(X)=RND(X):B(X)=RND(0):?A(X),B(X):NEXT
30 GOSUB"SORT A":' Or: CSUB" SORT A"
40 GOSUB"PRINT A":' Could be GOSUB 140
50 GOSUB"SORT B"
60 GOSUB"PRINT B"
70 END
80 '
100 "SORT A":' Alternatively: JNAME"SORT A"
110 SCLEAR:KEY A(0):TAG B(0):SORT 11:RETURN
120 "SORT B"
130 SCLEAR:KEY B(0):TAG A(0):SORT 11:RETURN
140 "PRINT A"
150 FOR X=0 TO 11: PRINT A(X),B(X):NEXT:RETURN
160 "PRINT B"
170 FOR X=0 TO 11: PRINT B(X),A(X):NEXT:RETURN
```

This program loads arrays A() and B() with random numbers and then proceeds to sort them individually, first on A() with B() elements "tagging along", then on B() with A() as a TAG.

This statement is used to scroll the video screen right one column.

RIGHT	STATEMENT
-------	-----------

'RIGHT' scrolls the entire screen right by one character, clearing the left-most (0th) screen column.

This function parses the right-hand substring of a string.

<code>RIGHT\$(exp\$,exp1)</code>	FUNCTION
<code>exp\$</code>	- is any string expression.
<code>exp1</code>	- is the number of rightmost characters to obtain from the string.

`RIGHT$` takes a substring away from the right. For example:

```
RIGHT$("ABCDEF",3) = "DEF"  
RIGHT$("NE PLUS ULTRA",10) = "PLUS ULTRA"
```

Note that `MID$` can easily simulate `RIGHT$`. For example:

```
RIGHT$(exp$,exp) is equivalent to MID$(exp$,len(exp$)-exp+1)
```

assuming `len(exp$) >= exp`.

This statement is used to set the printer's right hand margin.

RMARGIN = exp	STATEMENT
exp	- is a numeric expression which evaluates to the range <2-255>.

This statement sets the right hand margin on your printed page. An automatic carriage return done when the number of characters printed is equal to the value specified as 'exp'.

This function obtains a random number.

RND(exp)	FUNCTION
exp	- is a numeric expression.

'RND' returns a pseudo-random number between 0 and .999999 if 'exp' = 0; otherwise it returns an integer between 1 and 'exp'. Note that a sequence of numbers produced by the above function is not truly random.

The 'RANDOM' statement can be used to reseed the random number generator, further increasing randomness (or initiating a predetermined sequence for repeatable conditions).

All calculations are done in double precision to assure high randomness and a very long repeat cycle (which will occur eventually). The method of linear congruence is used, as described by Knuth in the second volume of his "The Art of Computer Programming"; this method fulfills all the usual tests of randomness while retaining simplicity of calculation.

This statement is used to establish a rotation for the 'DRAW' statement.

ROT = exp8	STATEMENT
exp8	- is a numeric expression which evaluates to the range (0-255) signifying DRAW degrees

This statement will set the rotation offset for DRAW statements. The direction is stepped in units of 256/360 degrees counter clockwise with "up" being 0.

This function obtains the current row position of the cursor.

ROW(dummy exp)	FUNCTION
----------------	----------

'ROW' returns the row of the cursor; equal to  $\text{INT}((\text{CURSOR POSITION or CURLOC})/(\text{number of columns}))$ . For example,

```
10 PRINT@170,"Hi."  
20 A = ROW(0)
```

assigns a 3 to 'A'.

'RSET' and 'LSET' are used to set information into FIELDed string variables for use with random access files.

RSET var\$ = exp\$	STATEMENT
var\$	- is FIELDed string to which the information is to be added.
exp\$	- is the information to add.

'LSET' and 'RSET' are really just versions of 'MID\$ ='. Their main intended purpose is to set information into FIELDed string variables. FIELDed strings must point to a static memory location (in a file's record buffer).

For 'LSET', var\$ is overlaid starting at position 0 with exp\$, filling any remaining portion of var\$ with blanks (ASCII 32). For 'RSET', var\$ is overlaid with exp\$, measuring from the end of var\$, filling any remaining portion of var\$ with blanks (i.e. the information is "right justified").

A standard string assignment, such as A\$="MONDAY" places A\$'s data in the string storage area, which is constantly changing. LSET and RSET (and MID\$) directly alter existing a string variable's contents without changing the string's position in memory. The main difference between MID\$ and LSET/RSET is that the latter commands fill the remaining characters in the affected string with blanks, or CHR\$(32)'s.

Note that compiled LSET and RSET, as with interpretive Disk BASIC LSET/RSET commands, work on any string variable, not just FIELDed string variables.

Examples (in all examples A\$ is 10 chars long):

LSET A\$="HELLO":'	Now A\$="HELLO      "
LSET A\$="12345678912":'	Now A\$="1234567891"
RSET A\$="HELLO":'	A\$="      HELLO"
LSET A\$=MKD\$(1.2345#):'	Now first 8 bytes of A\$ contain the floating point double precision number 1.2345#

'RUN' will load a '/CMD' type program from from disk and then invoke it.

RUN"filespec\$"	STATEMENT
filespec\$ - designates the file to run.	

'RUN' loads and runs a machine language program from disk. It can be any executable program including another compiled program.

This is used to establish a scaling factor for the 'DRAW' statement.

SCALE = exp16	STATEMENT
exp16	- is a numeric expression which evaluates to the range (-32768 to 32767)

This command sets the scaling factor for DRAW commands. The scaling factor is measured in units of 1/256. Thus, a "SCALE = 256" is equal to a 1:1 size plot. A "SCALE = 128" would be half sized.

This statement is used to turn on a pixel.

SET(x,y)	STATEMENT
x	- is a numeric expression which evaluates to the range <0 - 127> for 64-column screens and <0 - 159> for 80-column screens.
y	- is a numeric expression which evaluates to the range <0 - 47> for 16-row screens and <0 - 71> for 24-row screens.

SET, RESET, and COMPL form the set of the single-pixel-affecting graphics commands. Note that screens that display 16 rows of 64 characters will display 72 rows by 160 columns of graphics pixels; screens that display 24 rows of 80 characters will display 72 rows by 160 columns of graphics pixels.

SET is a standard TRS-80 BASIC command that unconditionally turns on any selected graphics pixel on the TRS-80 screen. The RESET command turns a pixel OFF. The COMPL command complements a selected graphics pixel, turning it ON if it is OFF and vice versa. A function, POINT(x,y), which is related to the pixel graphics commands is discussed in the section on functions.

The following illustrates a brief example of these graphics commands:

```
5   Y=23:RANDOM:CLS
10  FOR X=0 TO 127
20  SET(X,Y)
30  Y=Y+SGN(RND(3)-2)
40  IF Y<0 THEN Y=0 ELSE IF Y>47 THEN Y=47
50  NEXT
60  FOR X=0 TO 127
70  COMPL(X,23):NEXT
80  FOR X=0 TO 127
90  RESET(X,23):NEXT
```

The program first plots a pseudo-"mountainous" profile on the screen, proceeds to "complement" all graphics dots down the middle of the screen, and finally resets all pixels through the middle of the screen.

This function obtains the sign of its argument.

```
SGN(exp)
```

```
exp      - is a numeric expression.
```

The 'SGN' function will return -1, 0, or +1 depending on the state of its argument.

SGN(exp) = -1 if exp < 0

SGN(exp) = 0 if exp =0

SGN(exp) = 1 if exp > 0

These statements are associated with the built-in array sort.

<code>SORT &lt;(flag),&gt; num</code>	STATEMENT
<code>SCLEAR</code>	STATEMENT
<code>KEY array(exp)</code>	STATEMENT
<code>TAG array(exp)</code>	STATEMENT
<code>array(exp)</code>	- is an array element which designates the key array for sorting purposes and the tag array for sorting purposes.
<code>num</code>	- is an integer numeric operand in the range (1 to 32767) which designates the number of elements to sort.
<code>flag</code>	- is a numeric expression, either 0 or 1, to specify ascending or descending sort, respectively. If 'flag' is omitted, SORT defaults to ascending.

The SORT statement provides an easy but relatively fast way to sort single dimension (such as A(100), not A(40,20)) arrays using up to 32 keys and 32 "tags". 'SCLEAR' is an important SORT initialization command which must precede your sorting specification commands.

A one-key sort is straightforward. The keyed array is sorted, either in the default (no flag specified) ascending order, or in (flag=1) descending order. The sort time is variable, depending on the sort data and its organization, but a typical sort time for 1000 strings is 15 seconds.

TAGs are arrays which "tag" along with their associated keys and play no part in SORTing. If A(0)=5, A(1)=2, and B(0)=1 and B(1)=2, then if a single key sort on A(0)-A(1) were done with B(0)-B(1) as a tag, then the final result would be: A(0)=2, A(1)=5, B(0)=2, B(1)=1. Array element B(0) was "linked" to A(0) and B(1) to A(1) in the sort.

Multi-key sorts are also pretty straightforward. If identical entries are encountered in the current-level key, then the next-level-keyed array is sorted on, unless there are no more keys. IMPORTANT: The LAST array KEYed is the MOST SIGNIFICANT ("primary level"). The FIRST array KEYed is the LEAST SIGNIFICANT. Arrays are KEYed in LEAST to MOST significant order.

If the entries are not identical in the current-level key, then all lower-level KEYed arrays are TAGged.

Multi-key sorting is demonstrated with the following sample sort data:

```
A(0) = 2      B(0) = 3
A(1) = 4      B(1) = 6
A(2) = 3      B(2) = 7
```

```
A(3) = 2      B(3) = 7
A(4) = 3      B(4) = 5
A(5) = 1      B(5) = 3
```

Assuming that these values have been assigned, then the following:

```
SCLEAR:KEY B(0),A(0):SORT 6
```

performs the desired sort. The arrays are then:

```
A(0) = 1      B(0) = 3
A(1) = 2      B(1) = 3
A(2) = 2      B(2) = 7
A(3) = 3      B(3) = 5
A(4) = 3      B(4) = 7
A(5) = 4      B(5) = 6
```

As you can observe, array B is not in directly sorted order. It is only within "subfields" of A, where the array elements are the same, such as A(1) and A(2), and A(3) and A(4), that B's element are internally sorted; B(1) and B(2), and B(3) and B(4). In all cases, array B "tagged" along with array A. The only real distinction between TAG and KEY is that a TAGged array will appear in arbitrary order within primary key "subfields".

The EnhComp SORT facility allows you to specify the first element of the array for sorting to be at any point in the array. This is done implicitly when an array is KEYed or TAGged for sorting.

Example Program:

```
10 CLEAR 1000:DIM A$(20)
20 FOR X=0 TO 20
30 FOR Y=1 TO RND(5)
40 A$(X)=A$(X)+CHR$(RND(26)+64)
50 NEXT Y:PRINT A$(X),:NEXT X
55 PRINT:PRINT
60 SCLEAR:KEY A$(0):SORT 21
70 FOR X=0 TO 20: ?A$(X),:NEXT
```

This simple program generates and prints 21 random (1-5 character) length strings, sorts them, and prints out the sorted list.

This function obtains the square root of its argument.

SQR(exp)	FUNCTION
exp	- is a numeric expression.

'SQR' returns the square root of a non-negative expression (negative square roots are undefined in real (e.g. BASIC) numbers.) For example, SQR(4) = 2, since  $2 * 2 = 4$ , and SQR(81) = 9, since  $9 * 9 = 81$ . Usually the result is NOT a neat integer, as with SQR(7) (= approx. 2.64575). A double precision expression will cause a double precision square root to be returned, accurate to at least 16 decimal digits.

This statement is used to terminate your program with a message and then return to DOS.

STOP	STATEMENT
------	-----------

STOP causes a transfer back to DOS via the @EXIT address similar to END. The distinction between END and STOP is that the latter prints '-STOP-' <CR> and the current source line number (if available) before ENDing the program.

This function converts a numeric expression to an ASCII decimal string.

STR\$(exp)	FUNCTION
exp	- is any numeric expression

STR\$ is used to expand a binary number into its ASCII decimal equivalent. For example:

```
STR$(1.2+4.5)=" 5.7"
```

Notice the leading blank appearing in the string. The converted strings of all non-negative expressions will have such a leading blank. Negative expressions have a minus sign, "-", instead of a space.

This function generates a repeated character string.

STRING\$(exp1,exp2)	FUNCTION
STRING\$(exp1,"char")	FUNCTION
exp1	- is equal to the desired string length.
exp2	- is equal to a code in the range <0-255>
"char"	- is a single character.

STRING\$ is a convenient way to make long strings of the same selected character. For example:

```
STRING$(10,45) = "-----"  
STRING$(5,".") = "....."
```

'SWAP' is used to exchange the contents of two similarly typed variables.

SWAP var1,var2	STATEMENT
var	- is any variable

SWAP exchanges the values of two variables of the same type. If A\$="FIRST" and B\$="SECOND" then SWAP A\$,B\$ leaves A\$ with "SECOND" and B\$ with "FIRST".

This statement is used to set the video screen print zones.

SZONE(pos 1,...,pos n)	STATEMENT
SZONE(*)	
pos	- is a numeric expression between <0 and 63> which designates screen tab positions.

SZONE sets up default TAB positions for the "," modifier in PRINT statements and (equivalently) TAB(255) statements. SZONE(\*) clears all print stops. See program below for sample SZONE usage.

```
10 SZONE(*): 'Clear all tab stops
12 '
15 'Set up TAB stops in multiples of 8 spaces
17 '
20 FOR X=0 TO 63 STEP 8:SZONE(X):NEXT
30 FOR X=0 TO 30:PRINT X,:NEXT:' Could be PRINT X TAB(255)...
```

Once line 20 sets up stops, line 30 sample prints 0 through 30 showing the new tab stop intervals.

This function obtains the trigonometric tangent of its argument.

TAN(exp)	FUNCTION
exp	- is a numeric expression in radian measure.

TAN returns the radian degree tangent of an expression, mathematically equivalent to  $\text{SIN}(\text{exp})/\text{COS}(\text{exp})$ . It will return a double precision value if given one.

This function returns the system time as a string.

TIME\$	There is no operand	FUNCTION
--------	---------------------	----------

The system time is returned as an eight-character string of the form,  
HH:MM:SS.

These statements are used to provide for runtime program trace information.

```
TROFF
```

```
TRON
```

TRON acts similarly to interpretive BASIC TRON. It prints source line numbers (if available) after each statement is executed, not at just at the beginning of a source line. TROFF turns program trace off.

This function obtains the type code of its argument.

TYPE(exp)	FUNCTION
exp - is a numeric or string expression.	

'TYPE' returns the variable type code of the expression. These type codes are as follows:

Variable Type	Code
Integer	1
Single precision	2
double precision	4
string	3

Arrays are slightly more complex. The type is equal to:

$$128 + (16 * \text{dimension \#}) + \text{vartype}$$

where vartype is one of the standard variable type codes listed above. So,  $\text{TYPE}(A\$(0)) = 128 + 16 * 1 + 3 = 147$ . Note that the array index ('0') is arbitrary; it need only be within the dimensioned range.

This statement is used to scroll the video screen up one line.

UP	STATEMENT
----	-----------

'UP' scrolls the entire screen up by one line, clearing the bottom line. This is equivalent to the "standard" screen scroll.

This function is used to define formatted PRINT output.

```
USING format$;explist                                FUNCTION
format$ - is the format control string.
explist  - is the expression list.
```

The EnhComp 'USING' string function is derived from BASIC's, 'PRINT USING', which works equivalently compiled (= PRINT exp\$). The ability to store and manipulate 'USING' formatted data with string handling instructions makes this implementation much more versatile than the PRINT USING scheme.

USING's input is any mix of numeric and string expressions coupled with a string that controls the format of the output string. This format string is a concatenation of individual expression field specifiers. A complete list of field specifiers is given below.

USING processes the expressions one by one in a left to right manner, building up its output string as it processes each expression. For each expression processed, a field specifier in the format string expression is needed. Should the end of the format string be reached, the field specifier pointer is reset back to beginning of the format string. So:

```
USING "###.##";1.5555,2.6666,3.9999
```

causes the format string "###.##" to be "recycled" three times.

An error will occur if a string field specifier is tried on a numeric expression, and vice-versa.

Examples:

```
USING "###.##";3.157                = " 3.16"
USING "***###.##";1.45              = "***1.45"
USING "#####.#####";1.23456       = " 1.2346"
USING "$$###.##";19.95               = " $19.95"
USING "$$###.##";19.95               = "$19.95"
```

Assume X=7 in following examples:

```
USING "###.##";1.23,5.67,X*10        = " 1.23 5.67 70.00"
USING "!!!";"ALPHA","BETA","CANDY"  = "A B C"
USING "### ###.# ##.##";9.95,9.95,9.95 = " 10 10.0 9.95"
USING "##.## ##.# ";4.556,X*1.5,91.499 = "4.56 10.5 91.50"
USING "###.##-";15.69                = " 15.69"
USING "###.##-";-15.69                = " 15.69-"
```

The complete field specifier list is as follows:

Field Specifier List		
----- Numeric Formats -----		
Spec.	Purpose / definition	Example
#	One digit per # in field	###: 3 digits, round to nearest integer
.	Decimal point position	##.##: 2 digits to left of dec. point, round to nearest hundredth
+	Print leading/trailing sign (either + or -)	+###.## ###.##+
-	Print trailing sign only if negative	#####.-
**	Fill unused digits with asterisks instead of blanks	**###.##
\$\$	Put dollar sign at immediate left of number	\$\$\$\$###.##
**\$	Dollars sign at left of number and unused digits filled with asterisks	**\$###.##
^^^^	Format output in scientific notation	###.####
----- String Formats -----		
Spec.	Description	Example
!	First character of string expression	!";"ABC" = "A"
%blanks%	Include 2+# of blanks length substring of string expression	"% %";"ABCDE" = "ABC"

This function obtains the numeric value of the decimal number encoded in its string argument.

VAL(exp\$)	FUNCTION
exp\$	- is a string expression.

VAL converts an ASCII encoded decimal number to binary floating point or integer numeric format. For example:

```
A$ = "1.234": B$ = "4.5555555#": C$ = "156"  
A = VAL(A$): B = VAL(B$): C = VAL(C$)
```

sets 'A' equal to 1.234, 'B' equal to 4.55555 (truncated down to single precision from double precision), and 'C' equal to 156.

This function obtains the absolute memory address of its argument.

<pre>VARPTR(var)                                FUNCTION var      - is any numeric or string variable or array            element.</pre>
--

'VARPTR' is used to directly access variable data stored in memory. It returns the address of the first byte of a variable's contents. For example, supposing that the "LEN" function didn't exist. Then:

```
10  A$ = "ENHCOMP"
20  PRINT "LEN(A$) = ";!SLEN(A$)
30  END
100 FUNCTION SLEN(T$)
110 RETURN WPEEK(VARPTR(T$))
120 ENDFUNC
```

is a good example of creating a new function to fit a need (if LEN wasn't already supported). Note the use of the exclamation point which precedes the function's invocation. This is required by EnhComp for user defined functions and is explained in the section on FUNCTION-ENDFUNC.

VARPTR(T\$) returns the address to the start of T\$'s control block; which is in the form: LB LEN HB LEN LB PNTR HB PNTR. WPEEK(VARPTR(T\$)) returns the entire 16 bit length -- without WPEEK, it would be: PEEK(VARPTR(T\$)) + 256 \* PEEK(VARPTR(T\$)+1), considerably nastier.

A variation on the use of VARPTR is the use of an array's name without a subscript to return the address of the array's Data Control Block (DCB). This is denoted as:

<pre>arrayname()</pre>
------------------------

Arrayname() returns the address of the array's DCB.

For example:

```
TRIALS(), A(), ....
```

See the "Technical Section" for details on Data Control Block formats.

This function will wait for a keyboard entry and return the value of the key which is pressed.

WINKEY\$	FUNCTION
----------	----------

INKEY\$ returns the last key pressed. WINKEY\$ waits for a key to be pressed and then returns it as INKEY\$, a one character string.

Example Program:

```
10 PRINT"Press any KEY to continue, <ENTER> to loop"  
20 A$=WINKEY$:IF A$=CHR$(13) THEN 10  
30 PRINT"Exiting program"  
40 END
```

This function obtains the two-byte "word" stored at the specified memory address.

WPEEK(exp16)	FUNCTION
exp16	- represents a memory address in the range <-32768 to 32767>.

'WPEEK' effectively "peeks" two bytes at a time, forming a word in standard CPU format. The precise formula is:

$$\text{WPEEK}(\text{exp}) = \text{PEEK}(\text{exp}) + 256 * \text{PEEK}(\text{exp}+1)$$

WPEEK is useful for getting 16-bit memory addresses. For example, on the TRS-80 Model I/III:

```
V = WPEEK(&H401E)
```

assigns V to the memory address of the screen character print driver routine.

The corresponding poking statement, 'WPOKE', is described elsewhere in this manual.

This statement is used to poke a word into a memory location.

WPOKE exp16,exp16	STATEMENT
exp16	- specifies a memory address in the range <-32768 to 32767>.

WPOKE allows direct modification of any RAM location in memory. WPOKE "pokes" two bytes at a time in conventional low order/high order format into the specified address.

The 'XFIELD' statement is used to assign the segments of a type "X" file record buffer to strings.

XFIELD blknum,var,(exp)var\$,...	STATEMENT
blknum	- is file control block number, 1-15.
var	- is any non-string variable.
exp	- is the maximum length of the following string variable, var\$.
var\$	- is any string variable.

XFIELD is used to define the record structure of "X" type files. It fields the record buffer into segments accessible by string variables, providing a means to read and write information in an orderly manner from or to any record in the file.

For the variables specified in the variable list, integers take 2 bytes to store, single precision 4 bytes, double precision 8 bytes, and strings take the specified maximum length (given in the expression in parentheses preceding the string variable name) plus 2 bytes for the string length.

One advantage of using the extended file format is that the string length is saved at the time of the write and a subsequent 'GET' of that record will restore the string of the same length. This is unlike conventional FIELDS which pad unused characters with blanks. Note that if the string length exceeds the maximum given by 'exp', only the maximum number of characters in the string will be saved; all characters past that point will not be saved to the file.

The maximum record permissible in XFIELDed type files is 32767. Here is a sample XFIELD statement:

```
XFIELD 2,A%,B#,(16)INV$
```

Any subsequent 'PUT' statements (PUT bufnum,recnum) will write the current value of the variables A%, B#, and INV\$ into the specified record.

## 5 Technical Information

### 5.1 Variable names

Variable names are limited to the character set <A-Z>, <0-9>, and <@>. The first letter of the name must be an alphabetic character, <A-Z>. Variable names can be any length and are unique for their entire length. Thus, the following are all distinct variables: ABC, ABCDEF, AB123.

The only restrictions on variable names are that you cannot use the name of a BASIC STATEMENT or FUNCTION as the name of a variable. The BASIC STATEMENT names and FUNCTION names may appear as substrings of a variable name. This is permitted since all variable names must be separated from the "text" adjacent to the name by either a <SPACE> or a character not permitted as a variable name (i.e., ";", ":", etc).

### 5.2 Variable TYPE designations

As is standard with versions of Microsoft BASIC, the following characters can be used as a variable name suffix to designate the variable as being of the specific type identified.

Type Char	Variable Type Identified
%	Integer variable
!	single precision floating point variable
#	double precision floating point variable
\$	string variable

Variables may also be declared as being of a designated type by belonging to the operand class of a DEFINT, DEFSNG, DEFDBL, or DEFSTR statement.

### 5.3 Variable storage format

The following information describes the control block of arrays and the data storage format of the four supported variable types. A pointer to the control block (for arrays) or the data area (for scalars) is returned by the VARPTR function or its array counterpart, "arrayname()".

Array Data Control Block	Description of contents
DCB+0	number of dimensions
DCB+1	array type: 1=integer, 2=single prec, 3=string, 4=double precision.
DCB+2&3	Pointer to data area
DCB+4&5	Number of data entries
DCB+6&7 on up	size of each dimension
Integer Storage Format	Description of contents
LSB HSB	Value of the integer, 2-bytes
Single Precision Format	Description of contents
LSB MSB HSB EXP	Value of the single, 4-bytes
Double Precision Format	Description of contents
LSB MSB ... MSB HSB EXP	Value of the double, 8-bytes

<b>String Control Block</b>	<b>Description of contents</b>
DCB+0&1 (LSB MSB)	Length of string
DCB+2&3 (LSB MSB)	Pointer to the stored string

#### 5.4 Precision of math library

The math library supports operations using integers, single precision floating point variables and numbers, and double precision floating point variables and numbers. All supplied functions support both single and double precision arguments. This means that the result of functions such as LOG, EXP, COS, etc., is the precision of the argument used (single or double).

The range and precision of the three numeric types is as follows:

<b>number type</b>	<b>range</b>	<b>precision</b>
integer	-32768 to 32767	5 digits
single prec	-1.7e+38 to 1.7e+38	6-7 digits
double prec	-1.7d+38 to 1.7d+38	15-16 digits

#### 5.5 File buffer allocation

For each file buffer designated via the ALLOCATE statement, 592 bytes of memory will be provided. This memory is utilized as follows:

<b>Buffer offset</b>	<b>Intended use</b>
0	File type: "X", "I", "O", or "R" ("E" is converted to "O")
1	Record length of non-"X" file modes
2- 3	Record number of last PUT or GET
4	Unused
5	Internal buffer offset
6- 7	Unused
8- 9	Record length of "X" file mode
10- 11	Pointer to XFIELD data if "X" file mode
12- 13	Last file record number accessed
14	CLOSE flush flag ( <>0 = flush )
15	Unused
16- 79	System's File Control Block
80-335	File's 256-byte I/O buffer
336-592	File's user record buffer

## 5.6 Support Subroutine Descriptions

The most commonly used routines in a compiled program are in the library SUPPORT/DAT file; when required, individual support subroutines are appended onto a compiled program as needed. The routines extracted from the library and compiled into your program are identified during compilation by the numbers following the message:

### APPENDING SUPPORT SUBS

The following list notes the general function of each support subroutine. This list is provided only to help you in understanding the subroutine numbers which follow the above stated message. It is beyond the scope of this manual to provide detailed instructions on interfacing to these routines at the assembly language level.

- 000 - I/O, Interpret code stream, error trapping.
- 001 - POP stacked operands and set up for math routines.
- 002 - Floating point addition.
- 003 - Print evaluated expression.
- 004 - POP operand and place in the math memory accumulator.
- 005 - Floating point multiplication.
- 006 - Floating point division.
- 007 - Floating point subtraction.
- 008 - Arithmetic OR (integers).
- 009 - Arithmetic AND (integers).
- 010 - Compare the last two stacked operands for less than.
- 011 - Compare the last two stacked operands for greater than.
- 012 - Compare the last two stacked operands for equality.
- 013 - Arithmetic XOR (integers).
- 014 - Convert the word on the stack to an integer number.
- 015 - Interface to the @DATE and @TIME DOS functions.
- 016 - Load the following string literal onto the string stack.
- 017 - This performs the NEXT command of BASIC.
- 018 - Specified variable read from current DATA statement.
- 019 - The two topmost strings on the string stack are concatenated.
- 020 - "MID\$( exp\$, A, B)".
- 021 - Load the following string variable onto the string stack.
- 022 - Transfers stacked string to string variable.
- 023 - Handles "ON exp GOTO/GOSUB".
- 024 - Allocate temporary string space.
- 025 - Check the stack pointer for SP < (PRGTOP)+256.
- 026 - Test exp1\$ <> exp2\$.
- 027 - "RIGHT\$(exp\$,exp)".
- 028 - "LEFT\$(exp\$,exp)".
- 029 - "STRING\$(exp1,exp2)".
- 030 - "STRING\$(exp,exp\$)".
- 031 - "CHR\$".
- 032 - "INKEY\$"
- 033 - ">=", numeric
- 034 - "<=", numeric
- 035 - "=", string
- 036 - ">", string
- 037 - "<", string
- 038 - ">=", string
- 039 - "<=", string
- 040 - "LEN", numeric

041 - Resolve array varptr.  
042 - DIMension an array.  
043 - "INPUT" accessory subroutine.  
044 - "LINEINPUT" accessory subroutine.  
045 - Performs "TAB(n)".  
046 - Transfer resident math RAM accumulator to stack.  
047 - Prints the integer number contained in HL.  
048 - CVD executor.  
049 - CVS executor.  
050 - CVI executor.  
051 - MKD\$ executor.  
052 - MKS\$ executor.  
053 - MKI\$ executor.  
054 - Handles "BIN\$(exp)".  
055 - Handles "HEX\$(exp)".  
056 - "<>" routine, numeric.  
057 - LSET executor.  
058 - RSET executor.  
059 - Handles "OPEN type\$,bufnum,filespec\$<,reclen>".  
060 - GET executor.  
061 - PUT executor.  
062 - unused.  
063 - unused.  
064 - unused.  
065 - Performs all graphics commands.  
066 - Handles "VAL(var\$)".  
067 - Handles "STR\$(exp)".  
068 - "USING" string function.  
069 - "WINKEY\$" function.  
070 - "INSTR" function.  
071 - "END" routine.  
072 - Miscellaneous I/O subroutines.  
073 - "PRINT#" setup.  
074 - "CLOSE" routine.  
075 - Reinitializes video output.  
076 - "LINEINPUT#" routine.  
077 - "LOF" routine.  
078 - "EOF" routine.  
079 - File manipulation: LOAD, RUN, KILL, EXISTS, SYSTEM  
080 - STOP executor.  
081 - "INPUT#" routine.  
082 - Sets up current buffer and associated variables.  
083 - "LOC" executor.  
084 - Resolves DCB pointer given a filespec\$ or beffer expression.  
085 - Handles "MID\$(var\$,startpos<,maxfill>) = exp\$".  
086 - "POSFIL" assertor subroutines.  
087 - SORT routine.  
088 - Performs "PRINT,"; effectively TAB(255).  
089 - Single/double precision math routines.  
090 - Handles "ERROR exp".  
091 - Pushes defined function/command local variables onto the stack.  
092 - Supports command/function.  
093 - Restores local variable values.  
094 - Handles "PRINT <CR>".  
095 - internal support code.  
096 - Handles "PRINT@".  
097 - Creates a clean string list entry.

- 098 - "USING" initialization.
- 099 - "USING" post processing.
- 100 - "FRE(var\$)" executor.
- 101 - "RANDOM" executor.
- 102 - "RANDOM exp" executor.
- 103 - "ROW" function executor.
- 104 - "ASC" function executor.
- 105 - "LPRINT" initialization.
- 106 - "SWAP" executor.
- 107 - "KEY" executor.
- 108 - "TAG" executor.
- 109 - "SCLEAR" executor.
- 110 - "INP" executor.
- 111 - "PEEK" executor.
- 112 - "WPEEK" executor.
- 113 - "CURLOC" executor.
- 114 - "POS" executor.
- 115 - "ABS" executor.
- 116 - "ATN" executor.
- 117 - "CDBL" executor.
- 118 - "CINT" executor.
- 119 - "COS" executor.
- 120 - "CSNG" executor.
- 121 - "ERL" executor.
- 122 - "ERR" executor.
- 123 - "EXP" executor.
- 124 - "FIX" executor.
- 125 - "INT" executor.
- 126 - "SZONE/PZONE" executor.
- 127 - "LOG" executor.
- 128 - "MEM" executor.
- 129 - "RND" executor.
- 130 - "SGN" executor.
- 131 - "SIN" executor.
- 132 - "SQR" executor.
- 133 - "TAN" executor.
- 134 - "UNTIL" executor.
- 135 - a Z-80 "RET" instruction.
- 136 - integer "LET".
- 137 - Handles "var1^var2".
- 138 - "NOT" executor.
- 139 - "BRL" executor.
- 140 - Negate the value contained in the math memory accumulator.
- 141 - "CLS" executor.
- 142-166 - Various routines which deal with floating point stack operations.
- 167 - unused.
- 168 - "ALLOCATE" executor.
- 169 - "FIELD" executor.
- 170 - "IF" executor.
- 171 - "XFIELD" executor.
- 172 - unused.
- 173 - "GOTO" executor.
- 174 - "GOSUB" executor.
- 175 - Load the "READ" pointer.
- 176 - "RETURN" executor.
- 177 - "POP" executor.
- 178 - Load BASIC line number with the following word.

179 - internal use.  
180 - "OUT" executor.  
181 - "DEC" an integer variable.  
182 - "DEC" an integer array element.  
183 - "INC" an integer variable.  
184 - "INC" an integer array element.  
185 - Handler for INVERT, LEFT, RIGHT, UP, and DOWN.  
186 - Handles setting of ROTation and SCALE.  
187 - unused.  
188 - used internally.  
189 - Handles TRON, TROFF, BRKON, and BRKOFF.  
190 - internal CINT.  
191 - strobes keyboard for <BREAK>; performs TRON display.  
192 - load integer variable to math memory accumulator.  
193 - load single precision variable to math memory accumulator.  
194 - load double precision variable to math memory accumulator.  
195 - zero the math memory accumulator.  
196 - load integer number to math memory accumulator.  
197 - load single precision number to math memory accumulator.  
198 - load double precision number to math memory accumulator.  
199 - load integer array element to math memory accumulator.  
200 - load single precision array element to math memory accumulator.  
201 - load double precision array element to math memory accumulator.  
202 - equate integer variables.  
203 - equate single precision variables.  
204 - equate double precision variables.  
205 - equate integer variable with integer array element.  
206 - equate single precision variable with integer array element.  
207 - equate double precision variable with integer array element.  
208 - equate integer array elements.  
209 - equate single precision array elements.  
210 - equate double precision array elements.  
211 - equate integer array element with integer variable.  
212 - equate single precision array element with integer variable.  
213 - equate double precision array element with integer variable.  
214 - load integer variable to stack.  
215 - load single precision variable to stack.  
216 - load double precision variable to stack.  
217 - numeric integer "LET".  
218 - numeric single precision "LET".  
219 - numeric double precision "LET".  
220 - load integer array element to stack.  
221 - load single precision array element to stack.  
222 - load double precision array element to stack.  
223 - integer array element "LET".  
224 - single precision array element "LET".  
225 - double precision array element "LET".  
226 - integer "FOR" initialization.  
227 - single precision "FOR" initialization.  
228 - double precision "FOR" initialization.  
229 - push current code pointer for "REPEAT".  
230 - Handles "POKE exp1,exp2".  
231 - Handles "WPOKE exp1,exp2".  
232 - Begin execution of Z-80 code.  
233-255 - unused.

## 6 EnhComp Z80 Assembler Introduction

EnhComp, on top of being of a full BASIC compiler, is also a full Z80 assembler, with special numeric functions to return the VARPTR of a BASIC variable and the absolute memory pointer to the beginning of any line. No list of Z80 instructions is given here. It is assumed that as an experienced Z80 programmer, you already have at least one such list.

### 6.1 Z80 Source Code Inclusion in Programs

Z80 assembly language can be inserted at any point in the source program. The Compiler Directive 'Z80-MODE' switches the language context to Z80 mode.

Essentially, in Z80 mode, standard Z80 mnemonics take the place of BASIC instructions. Most standard Z80 assembler pseudo-ops, such as DEFB, are supported. As with BASIC instructions, multiple statements can be placed on a single line, separated by ':'s. This is a typical example of a combination BASIC / Z80 program:

```
10 DEFINT X
20 FOR X=0 TO 255
30 GOSUB "SCREEN"
40 NEXT
50 END
55 '
60 Z80-MODE
70 "SCREEN"
80 LD HL,3C00H:LD DE,3C01H:LD BC,03FFH
90 LD A,(&(X)):LD (HL),A:LDIR
100 HIGH-MODE
105 '
110 PRINT@0,X:RETURN
```

This sample program fills the Model I or III screen memory with every ASCII code, with each ASCII code number printed in the upper left hand corner. Its speed is rather impressive for a "BASIC" program.

Line 60 switches the compilation context to Z80 AL  
Line 70 defines a label, 'SCREEN'.  
Lines 80-90 define the Z80 subroutine itself.  
Line 100 switches the compilation context back to BASIC.

## 6.2 Assembler Expression Evaluation

Expressions are evaluated algebraically. '4+START\*10H' is evaluated as 'START\*10H plus 4', not in the linear fashion of '(4+START)\*10H'.

The following table describes the available assembler binary operators in algebraic priority order (top to bottom = highest to lowest):

"<"	exp1 < exp2	Exp1 shifted left 'exp2' times
<td>exp1 &gt; exp2</td> <td>Exp2 shifted right 'exp2' times</td>	exp1 > exp2	Exp2 shifted right 'exp2' times
".MOD."	exp1.MOD.exp2	Integer remainder of exp1/exp2
"*"	exp1 * exp2	Product of exp1, exp2
"/"	exp1 / exp2	Quotient of exp1, exp2
"+"	exp1 + exp2	Sum of exp1, exp2
"-"	exp1 - exp2	Exp1 minus exp2
".OR."	exp1.OR.exp2	Bit logical "OR" of exp1, exp2
".AND."	exp1.AND.exp2	Bit logical "AND" of exp1, exp2
".XOR."	exp1.XOR.exp2	Bit logical "XOR" of exp1, exp2
Boolean Operators -- return -1 if true, else 0 ( 'if' = 'if and only if'. All have equivalent weights and less priority than any of the above operators)		
".EQ." or "=". "	exp1.EQ.exp2	TRUE if exp1 equals exp2
".NEQ." or ".<>."	exp1.NEQ.exp2	TRUE if exp1 DOES NOT equal exp2
".LT." or ".<."	exp1.LT.exp2	TRUE if exp1 less than exp2
".GT." or ".>."	exp1.GT.exp2	TRUE if exp1 greater than exp2
".LTEQ." or ".<=."	exp1.LTEQ.exp2	TRUE if exp1 is less than or equal to exp2
".GTEQ." or ".>=."	exp1.GTEQ.exp2	TRUE if exp1 is greater than or equal to exp2

The following table describes the allowable numeric operand bases:

No suffix:	Base 10 = Decimal = Regular number
"V" suffix:	Base 2 = Binary ex: 1011V = 11 decimal
"H" suffix:	Base 16 = Hexadecimal ex: 4000H = 16384 decimal
"O" suffix:	Base 8 = Octal ex: 500 = 40 decimal

The following table describes the assembler Pseudo-Ops supported:

DEFB / DEFM / DB / DM	exp8 or 'textstring' (multiple operands allowed: Define byte(s) separate with commas. Example: DB 'PLAYER 1',13)
DEFW / DW	expl6 <,expl6,...> Define word(s)
DEFS	expl6 Leave 'expl6' bytes untouched
DEFF	expl6 <,exp8> Fill 'expl6' bytes with 00H. Optionally fill with 'exp8' if given
ORG	expl6 and DISORG Start a separate machine language load block with starting load address given by expl6. The "current" load address is saved. DISORG terminates the separate load block and re-establishes the old program counter so that normal compilation can continue. NOTE: Only the last PC is saved; nested ORGs are NOT advised.
Ex.:	ORG 401EH:DW ALTVID:DISORG ' Re-vector video char. display routine

The following table defines the EnhComp support of non-standard Z80 assembler instructions.

DUPI operand	("operand = operand*2 + 1")
where operand is any of:	-- r8 (A,B,C,D,E,H,L) (HL), (IX+d), (IY+d)

## 7 Invoking the REF/CMD utility

The REF utility provides a printed reference of memory use for five aspects of your program: variables, user defined functions, user defined commands, symbols and labels, and source line numbers. The listings are generated from the reference data file created by the compiler when the "WD" compiler directive is invoked.

The general format of a REF/CMD invocation is:

```
REF filespec<,-V-L>

filespec      - is the reference data filespec.
                The extension defaults to '/DAT'.

-V            - optional switch to direct the REF
                output to the video screen.

-L            - optional switch to generate the
                symbol/label table. The default is
                to suppress the symbol/label table.
```

The two command switches, "-V" and "-L", are optional. If either or both is entered, a comma must immediately follow the reference filespec. The "-V" switch is used to have the reference output appear on the video screen instead of the printer. The "-L" switch is used to have the "symbol/label" table included in the reference output.

The following represents excerpts from a given reference report. Note that all tables are alphabetized for easy reference. The five possible reports will each start on a new page. The first report will list all BASIC variables, identify each variable as to its type, and then list the starting memory address used to store the variable's value. A sample report is:

CROSS REFERENCE REPORT using CHEBYCO:4, --- VARIABLE LIST page 1.1

! = SINGLE, % = INTEGER, # = DOUBLE, \$ = STRING

A! : 5FD1H	A\$ : 5F99H	A1# : 6055H	A2# : 604DH
A3# : 6045H	AP# : 5FD5H	B#(1#) : 5FA9H	BP# : 5FDDH
C#(1#) : 5FA1H	CN# : 60B9H	CP# : 5FE5H	CS# : 60C1H
H\$ : 5F9DH	I! : 60A1H	J! : 6029H	K! : 600DH
L! : 60A5H	N! : 6005H	N1! : 6039H	N2! : 6009H
NC! : 6065H	NT! : 6035H	P# : 6099H	PA#(1#) : 5FC9H
PA# : 60E1H	RHO# : 6091H	RT# : 60D9H	S# : 602DH
S1# : 60A9H	S2# : 60B1H	SF# : 5FF5H	SP# : 5FEDH
ST# : 603DH	SUM# : 6011H	T#(1#) : 5FC1H	T1#(1#) : 5FB1H
T1# : 60C9H	T2# : 60D1H	TN#(1#) : 5FB9H	W# : 5FFDH
X# : 6019H	X1# : 6069H	X2# : 6021H	X3# : 6071H
XA# : 6089H	XF# : 6079H	XG# : 6081H	Z1# : 605DH

The second report lists any functions which have been defined in your program. The type of the function is listed as well as the memory address of the function. This will look like the following:

USER DEFINED FUNCTION LIST ----- page 2.1

! = SINGLE, % = INTEGER, # = DOUBLE, \$ = STRING

N\$ : 5230H

The third report identifies any user-defined commands. It will list the command name followed by the memory address of the command. If your program has no user-defined commands, the report will look like the following:

USER DEFINED COMMAND LIST ----- page 3.1

NO USER DEFINED COMMANDS

If you specify the "-L" switch, then the fourth report will generate a table of all symbols and labels used in the program being referenced. This will include all global symbols of SUPPORT/DAT library routines as well. Thus, the normal mode of REF/CMD is to suppress this report. If you do request it, it's listing will be like the following (truncated for brevity):

SYMBOL/LABEL LIST ----- page 4.1

@@ALLOC = 65ECH	@@BRKVEC = 65EAH	@@BRL = 658AH
@@BUFADR = 65BDH	@@C8 = 7DFAH	@@CF = 87DFH
@@CLRNUM = 65E4H	@@CP = 87CDH	@@CT = 87CCH
@@CURBUF = 65DEH	@@DG = 87E1H	
@@DIGBUF = 7E06H	@@DIGPNT = 7E02H	
@@DPPNT = 7E04H	@@DRWRTE = 65D5H	
@@DTSINE = 87B9H	@@DX2SINE = 87B1H	
@@EDIT = 87C5H	@@EF = 87E2H	@@ENDJUMP = 65DBH
@@ERL = 65E1H	@@ERR = 65E3H	@@ERRVEC = 65DFH

SYMBOL/LABEL LIST con't ----- page 4.2

@SR34 = 7096H	@SR4 = 6903H	@SR45 = 736FH	@SR45A = 7374H
@SR46 = 73DAH	@SR47 = 740BH	@SR71 = 7467H	@SSPSV = 65F1H
@SSRVECTBL = 6127H		@SSUB = 76AAH	@START = 65EFH
@STEMPNT = 6597H		@STRCMP = 6E16H	
@STRCMP5 = 6E38H		@STRPNT = 6889H	@TCHK = 8517H
@TMERR = 6467H	@TRSTR = 6CA2H	@TRSTRL = 6CAAH	
@TSTLNE = 8AEDH	@WRCUR = 60EEH	@X2SINE = 8225H	@ZTOP = 7478H
SLPNT1 = 6F19H	SLPNT2 = 6F1BH		

The last table generated lists each BASIC source line number followed by the memory address of the compiled line. This looks like the following (again abbreviated for brevity):

SOURCE LINE ADDRESS LIST ----- page 5.1

00100 : 521DH	00110 : 5229H	00120 : 5240H	00130 : 5245H
00140 : 527EH	00150 : 52B3H	00160 : 52CEH	00170 : 52D3H
00180 : 5302H	00190 : 5338H	00200 : 536BH	00210 : 53A2H
00220 : 53A7H	00230 : 53ABH	00240 : 53DFH	00250 : 53F7H

MISOSYS Enhanced BASIC Compiler Development System  
Copyright 1986 Philip A. Oliver, All rights reserved

00260	: 5430H	00270	: 5467H	00280	: 549AH	00290	: 54DEH
00300	: 54EEH	00310	: 5511H	00320	: 5518H	00330	: 5557H
00340	: 5566H	00350	: 557AH	00360	: 559EH	00370	: 55ABH
01340	: 5E0EH	01350	: 5E16H	01360	: 5E1AH	01370	: 5E1EH
01380	: 5E22H	01390	: 5E35H	01400	: 5E5CH	01410	: 5E64H
01420	: 5E68H	01430	: 5E70H	01440	: 5E74H	01450	: 5EA0H
01460	: 5EB3H	01470	: 5ED7H	01480	: 5EF1H	01490	: 5F04H
01500	: 5F11H	01510	: 5F37H	01520	: 5F61H	01530	: 5F7AH
01540	: 5F89H	01550	: 5F91H				

## 8 Alphabetic Function and Statement Summaries

### 8.1 Alphabetic Statement Summary

ALLOCATE <exp>	Allocates <exp> file buffers
BKOFF	Disable BREAK key
BKON	Enable BREAK key
COMPL(x,y)	Complement graphics pixel at (x,y); if pixel SET then RESET it, otherwise SET it
CLEAR <exp>	Set aside <exp> bytes for string storage; Zero / clear variables
CLS	Clear screen, home cursor
CLOSE n	Closes file buffer n; if no parameter, closes all open files
COMMAND name...	Mechanism to define start of user command
CSUB "label"	Makes a call to the specified label (must be a string literal)
DATA list	Define a list of data
DEC intvar	Decrement integer variable by one
DEFFN name	Single line user defined function
DEFDBL varnames	Variables included in the list will default to double precision if their types are otherwise unspecified
DEFINT varnames	Same as DEFDBL, except causes a default to integer
DEFSNG varnames	Same as DEFDBL, except causes a default to single precision
DEFSTR varnames	Same as DEFDBL, except causes a default to string type
DIM a1,a2,...	Dimension specified arrays
DOWN	Scroll entire screen down by one line
DRAW param	Using integer array as controller, SET, RESET, or "COMPL" (complement) turtle graphics on screen
ELSE	Defines default branch location if IF expression false
END	Stop program execution
ENDCOM	Specify end of user defined command definition
ENDFUNC	Specify end of multi-line user defined function definition
ENDIF	Terminate IF block
ERROR exp8	Force an "artificial" runtime error of error code "exp8"
FIELD param	Fields file buffer into blocks of strings
FOR (parameters)	Start a FOR-NEXT loop construct
FUNCTION name	Start multi-line user function definition
GET param	Reads one record from a file into its buffer
GOTO integerlit	Branch to program line
GOTO "label"	Branch to specified label
GOSUB integerlit	Call subroutine at program line
GOSUB "label"	Call subroutine starting at label
GTO "label"	Branch to specified label
INC intvar	Increments integer variable by one
INPUT var1,var2,...	Accept user keyboard input for variable values
INPUT#exp,var1,...	Assign variable(s) information read sequentially from specified ("exp") disk file
INVERT	Inverts all graphics on the screen
IF <exp> ...	Define beginning of conditional execution program block
JNAME "label"	Define label
KEY array(exp)	KEY array for SORTing purposes. KEYS specified in least to most significant sorting order. In other words, last array

	KEYed is primary sorting key. Multiple keys separated by commas allowed
KILL"filespec\$"	Delete specified filespec from disk
LEFT	Scroll entire screen left one character
LET var=exp	Set variable equal to algebraic or string expression
LINEINPUT ...	Assign string variable from verbatim keyboard input without default "? " prompt
LOAD"filespec\$"	Loads the machine language file specified by filespec\$
LPRINT list	Send list of information to printer
LSET var\$=exp\$	Sets var\$ = exp\$, with left justification
MID\$(var\$,exp1)=a\$	Overlay var\$ starting at position exp1 with the string expression 'exp\$'
MID\$(v\$,e1,e2)=exp\$	Overlay var\$ starting at position exp1 with 'exp\$' for a maximum of exp2 characters
NEXT v1,v2,...	Define end of FOR-NEXT loop
OPEN"parameters	Opens a file using the specified buffer #
ON BREAK GOTO addr	Causes branch to specified line or label if BREAK key hit and break scan active (BKON mode)
ON ERROR GOTO addr	Causes a branch to the specified line or label if (runtime) error occurs
ON exp GOTO list	Using expression, jumps to specified # in list
ON exp GOSUB list	Using expression, jumps to specified # in list
OUT exp1, exp2	Send exp2 out to port exp1
PAINT(x,y),paint	Color a bounded shape
PLOT param	Plots a line or a box on the screen
POKE exp1, exp2	Load memlocation exp1 with exp2
POP	Delete last GOSUB
POSFIL(#b,rec,ofs)	Position to specified point in sequential file. Functional with both "O" and "I" type files
PRINT list	Output list of information to specified device
PZONE(pos,pos,...)	Define printer TAB stops
PZONE(*)	Clear all printer TAB stops
PUT param	Writes the buffer contents to a file
RANDOM	Initializes the random # generator
RDGOTO addr	Positions DATA pointer to specified line # or label
RDGTO "label"	Positions DATA pointer to specified label
READ list	Reads a list of variables from DATA statements
REM or '	Define a remark
REPEAT	Define beginning of REPEAT/UNTIL construct
RESTORE	Restores DATA pointer to first data statement
RESUME line #	Used at the conclusion of an error trapping routine to jump to the specified line #
RETURN	Return from subroutine
RESET(x,y)	Reset graphics point at x,y
RIGHT	Scroll entire screen right one character
RSET var\$=exp\$	Sets var\$ = exp\$, with right justification
ROT=exp8	Set rotation offset (in 256 degree units) for subsequent DRAW statement executions
RUN"filespec\$"	Loads and executes the machine language program specified by filespec\$
SCALE=exp16	Set scalar line multiplier (in 1/256 units) for subsequent DRAWS. For example, SCALE=128 makes DRAW figures half their unscaled size
SCLEAR	Important initialization command for SORT. Use before any KEYing/TAGing done
SET(x,y)	Set graphics point at x,y

SORT exp	Ascending SORT of KEYed and TAGed arrays. Sort 'exp' number of elements
SORT(exp1),exp2	Ascending SORT if exp1=0, descending if exp1=1. Exp2 is number of elements to sort
STOP	Stops execution of the program and prints source line # if available
SYSTEM"command"	Invoke a DOS command string
SZONE(pos,pos,...)	Define screen TAB stops
SZONE(*)	Clear all screen TAB stops
SWAP var1,var2	Exchanges var1 and var2's values
TAG array(exp)	TAG array for SORTing purposes
THEN ...	Defines branch location for true IF expression
TROFF	Turn program trace OFF
TRON	Turn program trace ON
UNTIL exp	Defines end of REPEAT/UNTIL construct. Program execution branches back to last executed REPEAT if exp <> 0
UP	Scroll entire screen up by one line ("conventional" scroll)
WPOKE addr,exp	Does two byte poke of exp at addr

### 8.2 Alphabetic String Function Summary

BIN\$(exp16)	Convert 'exp' to 16 digit base 2 representation
CHR\$(exp8)	Convert 'exp8' to one byte string
HEX\$(exp16)	Convert 'exp16' to 4 digit hexadecimal representation
INKEY\$	Last key pressed on keyboard
LEFT\$(exp\$,exp)	Return 'exp' left most characters in exp\$
MID\$(exp\$,exp1)	Return all of string at point 'exp1' on
MKD\$(exp)	Convert 'exp' to 8 byte string representing a double precision Floating Point number
MKI\$(exp)	Convert 'exp' to 2 byte string representing an integer #
MKS\$(exp)	Convert 'exp' to 4 byte string representing a single precision Floating Point number
RIGHT\$(exp\$,exp)	Return 'exp' right most characters in exp\$
STR\$(exp)	Return ASCII DECIMAL equivalent of 'exp'
STRING\$(exp1,exp2)	Return 'exp1' long string of 'exp2' characters
STRING\$(exp1,exp\$)	Return 'exp1' long string of ASC(exp\$) characters
USING fmt\$;vlist	Return string using varlist, formatting determined by 'format\$'. Takes the place of the PRINT USING ... feature in interpretive BASIC. Performs equivalently
WINKEY\$	Wait for key and then return as one char string

### 8.3 Alphabetic Function Summary

&Bd0...d15	Accept digits in base 2 representation
&Hdddd	Accept digits in base 16 representation
&Odddd	Accept digits in base 8 representation
ABS(exp)	Returns the absolute value of the expression
ADDRESS("label")	Absolute memory address of 'label'
ADDRESS(line #)	Absolute memory address of line #
ASC(exp\$)	Returns the ASCII numeric code of the first byte of the string expression
ATN(exp)	Returns the arctangent (in radians) of the expression
CDBL(exp)	Converts expression to a double precision value
CINT(exp)	Converts expression to an integer value
COS(exp)	Returns the radian cosine of expression
CSNG(exp)	Converts expression to a single precision value

CURLOC	Current cursor position (0-1023)
CVD(exp\$)	Directly copies 8 byte string to a double precision numeric expression
CVI(exp\$)	Directly copies 2 byte string to an integer expression
CVS(exp\$)	Directly copies 4 byte string to a single precision expression
EOF(bufnum)	Returns '-1' if at end of specified sequential input file, '0' otherwise
ERL	Line # of the latest error
ERR	Code of the latest error
EXISTS(filespec\$)	Returns -1 if filespec\$ exists.
EXP(exp)	Returns the natural antilog of expression
FIX(exp)	Returns the integer value of the expression
FRE(exp)	Returns amount of free string space (or MEM if exp = 0)
INP(exp)	Returns eight bit value read from port 'exp'
INT(exp)	Return greatest integer less than 'exp'
INSTR(exp1\$,exp2\$)	Returns '0' if exp1\$ does not contain exp2\$, else returns the position of 'exp2\$'s first occurrence in exp1\$.
INSTR(e1,e1\$,e2\$)	Start search for exp2\$ at 'exp1'th character in exp1\$. INSTR(1,exp1\$,exp2) = INSTR(exp1\$,exp2\$)
LEN(exp\$)	Length of 'exp\$'
LOC(bufnum)	Returns last record accessed in specified random file
LOF(bufnum)	Returns number of records in specified file
LOG(exp)	Natural log of 'exp'
MEM	Amount of free memory
PEEK(exp16)	Eight bit contents of memory address 'exp16'
POINT(x,y)	Returns -1 if specified point is SET
POS(dummy)	Intra-line cursor position
RND(exp)	Returns a random # between 1 and exp
ROW(dummy)	Cursor row #
SGN(exp)	Signum function (1 if exp>0, 0 if exp=0, -1 if exp<0)
SIN(exp)	Returns radian sine of 'exp'
SQR(exp)	Returns square root of 'exp'
TAN(exp)	Returns radian tangent of 'exp'
TYPE(var)	Returns variable type of 'exp'
VAL(exp\$)	Changes ASCII DECIMAL string to internal numeric binary storage format
VARPTR(varname)	Absolute memory location of the specified variable or array element
WPEEK(addr) array()	Returns two byte contents (addr) + 256(addr+1) Address of the DCB of the specified array. Example: PRINT HITS() prints the address of the DCB of array HITS. See Technical Section for DCB breakdown

#### 8.4 Table of Numeric Operators

"↑"	A↑B	A to the Bth power
"*"	A*B	A multiplied by B
"/"	A/B	A divided by B
"+"	A+B	A plus B
"-"	A-B	A minus B
Boolean operators (-1 if true, else 0)		
"="	A=B	If A EQUALS B
"<"	A<B	If A is LESS THAN B
">"	A>B	If A is GREATER THAN B
"<>"	A<>B	If A DOES NOT EQUAL B
"<=" or "=<"	A<=B	If A LESS THAN OR EQUAL TO B
">=" or "=>"	A>=B	If A GREATER THAN OR EQUAL TO B
Logical BIT-WISE operators		
"AND"	A AND B	A logically 'AND'ed with B
"OR"	A OR B	A Logically 'OR'ed with B
"XOR"	A XOR B	A logically 'XOR'ed with B

#### 8.5 Table of String Operators

Comparisons are done on a character by character basis. They return numeric boolean values: -1 if true, 0 otherwise.

"="	A\$=B\$	A\$,B\$ precise equivalence check
"<"	A\$<B\$	A\$ alphabetically/ascii-ly less than B\$
">"	A\$>B\$	A\$ alphabetically greater than B\$
"<="	A\$<=B\$	A\$ alphabetically less than or equal to B\$
">="	A\$>=B\$	A\$ alphabetically greater than or equal to B\$
"<>"	A\$<>B\$	A\$ is not equal to B\$

## 8.6 Table of Compiler Errors

Error Code	Meaning
127	Dynamic data table overflow
128	"ENDIF" terminators missing
129	"ENDIF" without "IF"
130	Multiply defined User Function
131	Multiply defined Command Definition
132	Illegal label or symbol
133	Undefined label or symbol
134	Undefined User Command
135	Undefined User Function
136	Undefined line number
137	Expression type mismatch
138	Missing Operand
139	Syntax Error
140	Multiply defined symbol or label
141	Nested *GET/*INCLUDE file disallowed
192	(Z80) Expression error
193	(Z80) Relative branch out of range
194	(Z80) Operand field OVERFLOW

### 8.7 Table of run time Errors

Error Code	Meaning
0	Next without For
2	Syntax error
6	Out of Data
8	Illegal Function Call
10	Numeric Overflow/Underflow
12	Out of free memory
16	Array subscript out of dimensioned range
18	Attempt to re-dimension an array
20	Division by 0
24	Type mismatch
26	Out of string space
32-100	Special disk error; equal to DOS error code + 32
104	Illegal buffer #
106	File not in directory
108	Serial disk I/O attempted with "R" file mode
110	File already opened
122	Disk space full
128	Bad file name
130	GET or PUT attempted with non "R" file mode
134	Directory space full
136	Write protected diskette
138	File access denied due to password protection
162	Serial disk I/O attempted with non-256 LRL file
178	Attempt to open file with different LRL
241	SORT attempted without sort keys given
242	Too many sort keys or tags
254	Bad file mode (not "I", "O" or "R")