

# **LDOS™ & LS-DOS™**

## **BASIC Reference Manual**

**Cat. No. M-40-061**

**Copyright © 1992 MISOSYS, Inc.,  
All rights reserved**



**MISOSYS, Inc.**

**Copyright 1986, 1987, 1990 MISOSYS, Inc., All rights reserved**



# LDOS/LSDOS

## BASIC Reference Manual

*A reference to Interpreter BASIC  
and EnhComp Compiler BASIC*

**MISOSYS, Inc.**  
P. O. Box 239  
Sterling, VA 22170  
703-450-4181

*BASIC Reference Manual*: Copyright 1986-1992 MISOSYS, Inc.  
Combined Manual, First Edition (1992)

**All Rights Reserved.** No part of this reference manual, may be reproduced in whole or in part, either manually or automatically, by any means, including but not limited to the use of electronic, electromagnetic, xerographic, optical, network or BBS information and retrieval systems, without the express written consent of MISOSYS, Inc. Unauthorized reproduction and/or adaptation is a violation of United States Copyright laws and may subject the violator to civil penalties or criminal prosecution.

**Program License Agreement:** When accompanied by a Program disk or disks, this package is sold for use by the original purchaser on his/her machine only. *If being purchased by a company, school, or other entity with multiple machines, multiple users, or networked systems, single-copy purchases for multiple-use are not allowed or supported. Please write inquiring about our reasonable multiple-use and/or site-licenses or purchase extra copies from your dealer. The program may be copied, but for the purpose of archival copies only for the original purchaser's computer.*

Determination of suitability for any particular purpose whatsoever is the sole responsibility of the end-user. No warranties, expressed or implied, are given with regard to the suitability of this product for a particular purpose or application. Software is made available on an as-is basis, and MISOSYS, Inc. shall not be liable for any actual or consequential damages, whether real or alleged, arising from the use of this software.

EnhComp, LDOS, and LSDOS are trademarks of MISOSYS, Inc.  
MICROSOFT is a trademark of the Microsoft Corp.



<b>General Information .....</b>	<b>1</b>
Important note to be read first .....	1
Nomenclature used in this Reference Manual.....	1
About this manual.....	1
About BASIC variables.....	2
Variable names .....	2
Variable TYPE designations .....	3
Introduction to Interpreter BASIC.....	4
Invoking BASIC .....	5
Default Extensions .....	7
File Blocking .....	8
Program Protection .....	8
Single Stepping a BASIC Program .....	8
Tape Access .....	9
High speed Load and Save .....	9
Introduction to Compiler BASIC.....	10
Compiler BASIC Warranty .....	11
Compiler BASIC: General Information .....	11
Compiler BASIC Directives .....	12
High Level Statements .....	13
High Level Functions .....	13
Compiler String Functions .....	13
Editing Interpreter BASIC Programs.....	15
Loading and Saving BASIC Programs .....	15
Abbreviated Commands .....	19
Initiating automatic input mode .....	20
Deleting program lines .....	21
Editing existing BASIC program lines .....	21
Deleting a disk file .....	24
Listing programs .....	25
Clearing the resident program .....	26
Renaming a file .....	26
Copying, Moving, and Searching .....	26
BASIC Program Renumbering .....	29
BASIC Cross Reference Utility .....	31
Listing program variables .....	33
BASIC Compiler: Editing and Compiling .....	34
CED General Information .....	34
Invoking the REF/CMD utility.....	46

Compilation from CED Editor .....	51
Runtime errors .....	52
Command-line compiling .....	53
Compiler-generated line numbers .....	54
Compiler Directives .....	54
Compilation mode versus Interactive RUN mode .....	59
Independent use of compiled programs .....	59
 <b>BASIC Statements and Functions.....</b>	<b>61</b>
ABS .....	62
ADDRA .....	63
ALLOCATE .....	64
ASC .....	65
ATN .....	66
&B .....	67
BIN\$ .....	68
BKOFF, BKON .....	69
CALL .....	70
CDBL .....	71
CHAIN .....	72
CHR\$ .....	74
CINT .....	75
CLEAR .....	76
CLOSE .....	78
CLS .....	79
CMD .....	80
COMMAND .....	84
COMMON .....	87
COMPL .....	89
CONT .....	90
COS .....	91
CSNG .....	92
CURLOC .....	93
CVD .....	94
CVI .....	95
CVS .....	96
DATA .....	97
DATE\$ .....	99
DEC .....	100
DEFDBL, DEFINT, DEFSNG, DEFSTR .....	101
DEFFN .....	102

---

## Table of Contents

---

DEFUSR	104
DIM	105
DOWN	107
DRAW	108
END	110
EOF	111
ERASE	112
ERL	113
ERR	114
ERROR	115
ERRS\$	116
EXISTS	117
EXP	118
FIELD	119
FIX	122
FOR ... NEXT	123
FRE	125
FUNCTION	126
GET	129
GOSUB	130
GOTO	132
&H	133
HEX\$	134
IF THEN ELSE	135
INC	137
INKEY\$	138
INP	139
INPUT	140
INPUT#	141
INPUT\$	143
INPUT@	144
INSTR	145
INT	147
INVERT	148
JNAME	149
LEFT	151
LEFT\$	152
LEN	153
LET	154
LINEINPUT	155
LINEINPUT#	157
LINESPAGE	159

---

LMARGIN	160
LOAD	161
LOC	162
LOF	163
LOG	164
LPOS	165
LPRINT	166
LSET	167
MEM	170
MERGE	171
MID\$0=	173
MID\$	175
MKD\$	176
MKI\$	178
MKS\$	179
&O	180
OCT\$	181
ON exp	182
ON BREAK	183
ON ERROR	184
OPEN	185
OPTION base	192
OUT	193
PAGELEN	194
PAINT	195
PEEK	196
PLOT	197
POINT	199
POKE	200
POP	201
POS	202
POSFIL	203
PRINT	204
PRINT#	206
PRINT USING	213
PUT	217
PZONE	219
RANDOM	220
RDGOTO	221
READ	222
REM	223
REPEAT - UNTIL	224

---

## Table of Contents

---

RESET	227
RESTORE	229
RESUME	230
RETURN	231
RIGHT	232
RIGHT\$	233
RMARGIN	234
RND	235
ROT	236
ROW	237
RSET	238
RUN	239
SCALE	243
SET	244
SETEOF	245
SGN	248
SIN	249
SORT, SCLEAR, KEY, TAG	250
SPACE\$	255
SPC	256
SQR	257
STOP	258
STR\$	259
STRING\$	260
SWAP	261
SYSTEM	262
SZONE	264
TAN	265
TIME\$	266
TROFF, TRON	267
TYPE	268
UP	269
USING	270
USR	271
VAL	279
VARPTR	280
WAIT	282
WHILE WEND	283
WIDTH	284
WINKEY\$	285
WPEEK	286
WPOKE	287

WRITE	288
XFIELD	289

<b>Technical Information</b>	<b>291</b>
BASIC Statements	291
String functions	297
Numeric functions	298
Numeric BINARY operators	301
String operators	302
Variable storage format	302
Precision of math library	304
File buffer allocation	305
Compiler BASIC Support Subroutine Descriptions	306
Compiler BASIC Z80 Assembler	311
Z80 Source Code Inclusion in Programs	311
Access of BASIC variables and line numbers	312
Assembler Expression Evaluation	312
Operand Bases	314
Non-standard Z80 Instructions	314
Assembler Pseudo-OPs	315
Compiler BASIC error codes	316
Compile-time Errors	316
RUNTIME errors	317
Interpreter BASIC Error Codes	320
Error Definitions	320
Index	325

### General Information

#### Important note to be read first

Certain documentation pertaining to the programs referenced in this manual may be available after the *Reference Manual* has gone to press. If so, There may be a file named "README/TXT" on the DOS or program disk. Consult this file for details on additional support material and errata as this file will contain important information which may not appear in this printed documentation. You should read this file by issuing the command:

#### LIST README

#### Nomenclature used in this Reference Manual

Throughout this reference manual, illustrations of communications with the operating system and/or BASIC and you are presented in various forms. These are:

**EXAMPLE** font used to depict keyboard entries typed by you

**Message** font used to indicate a message displayed by BASIC

**screen** font used in presentation of display screens

**[optional]** square brackets surround optional keyboard entries

**on | off** a vertical bar is used to indicate either of two permissible entries, only one of which may be entered at a time.

#### About this manual

This *BASIC Reference Manual* is set up to be easily used. It is divided into several different sections. The first section is composed of general information about BASIC. It contains information specific to "Interpreter BASIC", common to both DOS version 5 and DOS version 6; it also contains information specific to a BASIC compiler called EnhComp, which is available from MISOSYS. Throughout this manual, "EnhComp" will be referred to as "Compiler BASIC". All BASIC commands which refer to the entry and editing of BASIC programs will be covered in this

section; thus, commands such as EDIT, LIST, and RENUM will be discussed here.

The second section contains reference material on all the BASIC statements and functions supported by the BASICs previously noted, exclusive of any editing commands. They will be listed in alphabetical order. Because differences in dialect occur across these implementations, those differences will be noted as they appear in the *Reference Manual*. **Please remember that not every command, statement, or function is supported by all implementations.**

This manual is written as a reference guide only. All commands will be explained in terms of the function which they serve. In no way will this manual serve as a tutorial on implementation of these commands. There are many such books currently on the market that deal with using a "Microsoft compatible" disk BASIC for generalized and specific applications. If you require tutorial aids for utilizing BASIC, contact your computer dealer or book store for a list of such material.

### About BASIC variables

#### Variable names

Variable names are limited to the character set <A-Z> and <0-9>; Compiler BASIC also allows <@>. The first letter of the name must be an alphabetic character, <A-Z>.

DOS 5 Interpreter BASIC variable names can be of any length; however, only the first two characters are significant. The restriction on variable names is that you cannot use the name of a BASIC Statement or Function as the name of a variable. Also, a reserved word (the name of a statement, operator, or function) is not permitted as a substring of a variable name.

Compiler BASIC, as well as DOS 6 Interpreter BASIC, permit long variable names (up to 40 characters in length), unique for their entire length, and allow reserved words (BASIC statement and function names) to be substrings of variable names. As such, it is required that all statement and function names, as well as variable names be delimited by either a <SPACE>, a special character not acceptable as a name (i.e. :,"\*+/-<>=), or an end of line. 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. Thus, the following are all distinct variables: ABC, ABCDEF, AB123.

### Variable TYPE designations

Without any overriding type declaration, either explicit or implicit, variables are of type single precision. As is standard with versions of Microsoft BASIC, the following characters can be used as a variable name suffix to explicitly 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. This is a declaration whereby the type is implicitly designated according to the first character of the name.

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. Interpreter BASIC supports string lengths from 0 to 255 characters. With Compiler BASIC, this length can be from 0 to 32767, a significant improvement over the 255 character limitation of interpretive BASICs.

## **Introduction to Interpreter BASIC**

Your computer contains two different types of memory: ROM (Read Only Memory) and RAM (Random Access Memory). ROM contains the routines necessary to get your computer started; ROM under DOS 5 may also contain a portion of the BASIC interpreter. This ROM BASIC allows you some capabilities of programming in the BASIC language. However, ROM BASIC does not allow you to interface with your disk drives when programming, and hence does not fully utilize your disk system.

The BASIC provided with your DOS is either an extension of ROM BASIC and resides in RAM, or is a complete BASIC interpreter. DOS version 5 BASIC utilizes commands found in ROM BASIC, and adds commands to ROM BASIC which will allow you to interface your BASIC programs with the disk operating system.

The disk-BASIC extension or complete "Interpreter" BASIC is contained in files provided on your DOS "Master Diskette". Included are a primary BASIC interpreter program and support utilities, some of which are present as overlays. They are:

<b>BASIC/CMD</b>	Disk Basic program.
<b>BASIC/HLP</b>	A file of HELP information for DOS version 5 BASIC
<b>BASIC/OV1</b>	This file contains the library command overlay segment of the DOS version 6 BASIC interpreter. It contains the CMD"N" renumber feature overlay used with DOS 5 BASIC.
<b>BASIC/OV2</b>	This file contains the routines for the DOS version 6 BASIC line copy, move, find, and search functions. For DOS version 5, it contains BASIC's cross reference CMD"X" feature.
<b>BASIC/OV3</b>	This overlay contains the BASIC handling of error display and CMD"O" sort routines for DOS version 5 BASIC.

### BASIC/OV4

A DOS version 5 BASIC overlay to dump a list of active variables

### Invoking BASIC

This is the syntax to be observed when invoking BASIC.

#### Interpreter BASIC

<b>BASIC [program/BAS] (F=n,M=n)</b>	6
<b>BASIC (B=sw,F=n,M=n,HIGH LOW,E=off) command</b>	5
<b>BASIC *</b>	5

**program** This may be the name of a BASIC program which will be loaded and RUN; a /BAS extension is required if the program filespec includes it.

**Files=** Optional parameter that specifies the maximum number of files BASIC will be able to access (1 to 15). If not specified, 3 is assumed.

**Mem=** Optional parameter to set the highest memory address to be used by BASIC. All memory above this address will be "protected". If not specified, all memory up to HIGH\$ will be available.

Note: "5" refers to DOS 5 BASIC; "6" refers to DOS 6 BASIC

**Additional DOS 5 Interpreter BASIC Parameters**

<b>*</b>	Used to re-enter BASIC with the program and the variables intact.
<b>Blk=</b>	Optional parameter that specifies Blocked file mode, either ON or OFF. ON is the default.
<b>Ext=</b>	Optional switch to turn off the default file extension "/BAS" used with the BASIC commands LOAD, RUN, MERGE and SAVE.
<b>HIGH or LOW</b>	Model III parameter that sets the cassette baud rate, either HIGH or LOW (HIGH=1500 and LOW=500). The default is HIGH. If HIGH is used, the HITAPE command must be issued prior to entering BASIC.
<b>command</b>	This may be any valid BASIC command which will execute immediately upon entering BASIC, such as RUN"MYPROG/BAS", AUTO100, etc

The "command" specification is also optional. If not specified, you will enter into BASIC, a welcome message will be displayed, and the BASIC Ready prompt will appear on the screen. The "Ready" prompt will indicate that BASIC is ready to accept any command that you wish to give it.

If you have rebooted the DOS 5 system, or have performed an exit from BASIC to the operating system (usually done by issuing a CMD"S" command), and wish to re-enter DOS version 5 BASIC, you may enter the command:

**BASIC \***

at the DOS Ready level. Doing so will cause BASIC to be re-entered, and any program that was resident in memory prior to performing the exit to the DOS Ready level will remain intact. Be aware of the fact that if **BASIC \*** is used to re-enter BASIC from the DOS Ready level, any commands which affect HIGH\$, or any commands that utilize memory (such as BACKUP and COPY) may cause your BASIC program to be overwritten

with other information. For this reason, **BASIC \*** should only be used as a last resort.

One of the following commands may be given if you wish to enter BASIC with two files open and have memory protected up to location 61440 (X'F000'). Also, you wish to have the program MYPROG/BAS loaded and RUN upon entering BASIC.

<b>BASIC (Files=2,Mem=61440,) RUN"MYPROG/BAS"</b>	<b>V5</b>
<b>BASIC (F=2,M=X'F000') RUN"MYPROG"</b>	<b>V5</b>
<b>BASIC MYPROG/BAS (F=2,M=X'F000')</b>	<b>V6</b>

Issuing either of the first two commands will produce the same results. The second command above uses the abbreviations "F" and "M" for "Files" and "Mem". Note that the extension for the program MYPROG/BAS need not be specified since EXT is ON. The third command is used for DOS version 6 and will automatically run "MYPROG/BAS". Also, realize that for any of the above commands, if HIGH\$ is lower than 61440 (X'F000'), an "Out of Memory" error will occur, and you will be returned to the DOS Ready prompt without entering BASIC.

### Default Extensions

DOS version 6 BASIC provides no default file extension for use with program management; however, it is recommended that you habitually use an extension of "/BAS" to clearly differentiate your BASIC programs from other files on your disks.

DOS version 5 BASIC allows you to utilize the default extension of /BAS when issuing the LOAD, RUN, MERGE and SAVE commands. If the EXT parameter is not turned OFF when entering BASIC, all filespecs used with the above commands that do not have extensions will be assigned the extension /BAS. If EXT is on and an extension is specified, the extension used in the filespec will override the default extension.

If EXT is ON and the file in question has no extension, it must be specified as "filename/" (i.e. the "/" will override the default /BAS). If the EXT parameter is turned OFF when entering BASIC, all file extensions will have to be specified.

### File Blocking

BASIC provides a Blocked file mode (which has often been misnamed Variable Length Files). This mode allows files with Logical Record Lengths (LRL) of less than 256 bytes to be created and accessed. Any record length from 1 to 256 bytes will be allowed, even if the record size is not evenly divisible into 256. Blocked file mode is optional with DOS version 5 BASIC; it is always in effect with DOS version 6 BASIC.

All blocking and de-blocking across "sector boundaries" will be performed by DOS. In this way, user records can span across sectors to provide maximum disk storage capacity. If the LRL is not specified when OPENing a Random file, "256" will be assumed. Note that an LRL of "0" will signify a 256 byte LRL.

If the Blocked file mode is ON, each file declared when entering BASIC will take 546 bytes of memory (564 for DOS version 6 BASIC). If the Blocked mode is OFF, each DOS version 5 file will take 290 bytes.

### Program Protection

DOS version 5 BASIC programs may be protected with an "Execute only" password. This means that the program may be RUN, but not LOADED, LISTed, LLISTed, or otherwise examined. Any attempt to break the program execution and examine the program will cause the program to be erased from memory, and the message "Protection has cleared memory" will be displayed. The DEBUGger will also be disabled during program execution.

### Single Stepping a BASIC Program

This DOS version 5 BASIC feature allows the BASIC programmer to step through each program statement singly, with a "HOLD" after each step. To invoke this feature simply do a normal pause (<SHIFT @>), which will cause BASIC to go into a wait state. While continuing to hold down the <SHIFT @> press the <SPACEBAR>, and the next BASIC statement will execute. After execution of that statement the computer will immediately go into its wait state again. Holding down the <SPACEBAR> will execute statements at the normal keyboard repeat rate. If you press any key without holding down the <SHIFT @>, normal program execution will resume. Note that this feature also functions when listing a program.

### Tape Access

Model I users need to disable the interrupts prior to performing tape I/O, and must re-establish them after the input/output has been performed. To disable the interrupts, use the BASIC command - CMD"T" -. To enable the interrupts, use the command - CMD"R" -. See the *BASIC Statements and Functions* Section for more information on these two commands.

Model III users need to do one of several things, depending on the type of tape involved. If you are dealing with a 500 baud tape, you will need to specify the "LOW" parameter when entering BASIC (Remember, if "HIGH" or "LOW" is not specified, the default will be HIGH). If you are dealing with a 1500 baud tape, you will need to establish the HITAPE utility. For more information on HITAPE, refer to the DOS manual.

### High speed Load and Save

When using the normal SAVE or LOAD program commands for tokenized (compressed) BASIC programs, disk I/O should be two to three times faster than standard Microsoft BASIC. Programs saved with the ASCII parameter will not enjoy this speed increase, either when saving or when loading.

### Introduction to Compiler BASIC

To begin with, the Compiler BASIC Development System comprises a minimum of five files. These are BC/CMD, CED/CMD, REF/CMD, S/CMD, and SUPPORT/DAT. These files are DOS-specific, i.e. there is a specific version for DOS 5 and another for DOS 6. Both versions are distributed on standard DOS 40-track double density data diskettes.

**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, Compiler BASIC 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 Compiler BASIC keyword tokenization, CED (through S/CMD) 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 Compiler BASIC system. SUPPORT/DAT must be available on one of your disks during every compile. Compilation will automatically be aborted if SUPPORT/DAT isn't 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.

Compiler BASIC 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 pretty complex. An experienced assembly language programmer can almost always produce more efficient code than a compiler, including the so-called optimizing compilers. Because a "core" of subroutines are included as needed, the size of relatively short Compiler BASIC 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.

### **Compiler BASIC Warranty**

The Publisher of Compiler BASIC makes no guarantee as to the fitness of Compiler BASIC, or programs generated by Compiler BASIC, for any particular use, nor do we assume any liability whatsoever for any damages that may arise directly or indirectly through the use of Compiler BASIC 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 Compiler BASIC you imply acceptance of these terms.

### **Compiler BASIC: General Information**

Compiler BASIC is a compiler, which differentiates it from 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 totally to “pseudo-code”, which is space efficient but slow. Compiler BASIC is a true compiler; it compiles directly to Z80 machine language; however, an internal pseudo-code is used to link program fragments with the system support library modules linked to a program and called as subroutines.

Compiler BASIC is unique for DOS. 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. Compiler BASIC, in fact, is not only a compiler, but a Z80 assembler that allows powerful algebraic expressions in source code statements, and takes advantage of the high level language/machine language intermix ability, with special functions that allow access to variable, line number, and label addresses.

Compiler BASIC is not guaranteed to translate your interpreted BASIC programs unmodified into machine language. However, any differences are slight and easily fixed to accommodate compilation. The large repertoire of commands and functions make it likely that you will be writing old programs over using these new features, rather than settling for the capabilities of the disk BASIC interpreters.

Compiler BASIC retains many of the “nice” features of interpreted BASIC that are excluded in other 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, up to fifteen dimensions [e.g. 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 not 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.

### Compiler BASIC 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 will be discussed in this section.

### 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" BASIC 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.

### **Compiler String Functions**

Compiler BASIC 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.

### Editing Interpreter BASIC Programs

Editing of Interpreter BASIC is performed, for the most part, while in the BASIC interpreter. Compiler BASIC editing is performed using the CED command. There are editing commands common to both Interpreter BASIC and Compiler BASIC; however, in order to avoid confusion, all Compiler BASIC editing facilities will be discussed via material referencing the CED command; the material will duplicate some material in this section.

### Loading and Saving BASIC Programs

#### LOAD

The LOAD command allows you to retrieve a BASIC program that has been stored on disk, and place it in the computer's memory so that it may be executed or edited. The syntax for the LOAD command is:

<b>LOAD"filespec\$"[,R]</b>	Statement
<b>filespec\$</b>	Designates the file to load; it may be a string constant or expression. If represented as a string constant, filespec must appear within quotes.
<b>R</b>	Is an option to cause the loaded program to be immediately RUN.

The "R" parameter is optional; if used, the program to be loaded will be executed after it is loaded, and all open files from a currently loaded BASIC program will remain open. Performing a LOAD without the "R" option will cause any open files to be closed.

Loading a program will always overwrite any program in memory with the program to be loaded. BASIC programs cannot be concatenated with the LOAD command (see "MERGE" for program concatenation). The LOAD command may be given from the BASIC Ready prompt, or can be issued from within a program. If issued from within a program, the program issuing the LOAD command will be overwritten by the program to be loaded, and execution will be terminated.

For example, after execution of the command **LOAD"MYPROG/BAS"** any program which was in memory will be replaced by the program **MYPROG/BAS**.

## **SAVE**

The **SAVE** command will allow you to save the program currently in memory to a disk file. This will allow you to store programs on disk for future use. The syntax for the **SAVE** command is:

**SAVE"filespec"[,A]**

Statement 5

**SAVE"filespec"[,A][,P]**

Statement 6

**filespec** Is the file specification you wish to assign to the program file. It may be represented as either a string constant or a string expression.

**A** An optional parameter to save the file in pure ASCII format. If not specified, the program will be saved "compressed" or tokenized format.

**P** An optional version 6 parameter that causes the file to be saved in an encoded format. Such programs are protected from listing or editing.

As BASIC programs are being written or edited, they are contained in the computer's memory. The **SAVE** command provides a way to save BASIC programs which are stored in memory out to a disk file, so that they may be referenced at some later time via the **LOAD** or **RUN** command.

When the **SAVE** command is given, one of two things will happen. If the **filespec** in the **SAVE** command represents a non-existing file, **SAVE** will create a file with the filename, extension, and password specified, and store in this file the BASIC program currently in memory. If the **filespec** in the **SAVE** command represents an already existing file, **SAVE** will overwrite the existing file with the program in memory.

When the "A" parameter is not specified in a **SAVE** command, the program in memory will be saved to a disk file in its compressed form (i.e.

token codes will be used to represent the BASIC commands and line numbers). If the "A" parameter is specified in a SAVE command, the program will be saved to the disk file in pure ASCII (e.g. the command PRINT will take up five bytes of disk storage, one byte for each character).

Note: When using the "A" parameter to save a program, no line in the program should exceed 240 characters in length. If a program is saved with the "A" parameter and a line in the program is longer than 240 characters, the program, when loaded, will load up to the line which is longer than 240 characters, and the rest of the program will be inaccessible. A Direct Statement in File error will also be generated.

It should be obvious that saving a program in ASCII will consume more disk space than saving the same program in compressed form, but there are certain situations where a program must be saved in ASCII. One case where you have to save a program in ASCII is if you wish to perform a MERGE of a BASIC program stored on disk with a program currently in memory. The program to be merged in from disk must have been saved in ASCII, or the merge will abort with an error.

The SAVE command may be given either from the BASIC Ready prompt, or may be incorporated as a command within a program. If used within a program, the program will SAVE itself, after which normal execution will continue.

Suppose you have keyed in a BASIC program, and wish to save this program out to a disk file. The drive you wish to store this file on is drive :1, and the name you wish to assign to this file is GOODPROG/BAS. One SAVE command that may be used to accomplish this might look like this:

**SAVE"GOODPROG/BAS:1"**

If you wish to save this program in ASCII, the following command could be used:

**FS\$="GOODPROG/BAS":SAVE FS\$,A**

Note in the above example that the filespec was represented as a string variable. Also note that the "A" parameter must appear as a literal constant, and cannot be expressed as a string expression.

### CLOAD and CSAVE

These statements are used to load or save a BASIC program stored on cassette tape. The syntax is:

#### DOS 5 Interpreter BASIC

**CLOAD[?] ["filename"]**

Statement

**CSAVE ["filename"]**

Statement

**filename** Is the single-character name used to identify the program file on tape. The name can be any alphanumeric character other than "". The name is mandatory for a CSAVE; if omitted for a CLOAD, the first program found will be loaded.

**?** Used to compare a program on tape with the one already loaded into the computer.



### Abbreviated Commands

A few of the BASIC commands may be represented as single characters. When using a single character command, the effect will be identical to using the entire word. This abbreviated form is only acceptable when typed on a command line, not in a program line or JCL file.

#### Interpreter BASIC Single-key edit commands

<↑>	Cursor up one line	
<↓>	Cursor down one line	
<←>	Cursor to first line	
<→>	Cursor to last line	
<.>	Display current line	
<,>	Edit current line	
<A> n1[,inc]	AUTO: AUTO line n1 with increment	
<C> n1,n2	Copy line n1 to line n2	
<D> n1[-n2]	DELETE: Delete line(s) n1 through n2	
<E> n1	EDIT: Edit line n1	
<F> Object	Find all object (line#, var, keyword)	6
<L> n1[-n2]	LIST: List line(s) n1 through n2	
<M> n1,n2	Move line n1 to be line n2	5
<M> n1,n2,n3	Move lines n1-n2 to be follow n3	6
<S> Object	Find next object (line#, var, keyword)	6

In the case of "A" for AUTO, "D" for DELETE, "E" for EDIT, and "L" for LIST, these commands work exactly as their full length counterparts, except that no space is necessary between the letter and the line numbers. For example, "L100-300" is the same as "LIST 100-300".

Under DOS version 6 BASIC, you may get "LIST OCATE", "EDIT", "DELETE", or "AUTO" at the beginning of some line in your BASIC program. This may occur when you are loading in a program from some other source in ASCII format. If the first word in the line is not a reserved keyword, and the first letter of the word is an "A", "D", "E", or "L", that first letter is expanded to "AUTO", "DELETE", "EDIT", or "LIST", respectively. The way to correct it is to edit the word.

The following six “immediate” key commands are implemented by pressing the indicated key as the first character in the command line. No carriage return is necessary; the indicated action will take place immediately. Note that any of the following single key commands must be the first character entered after the “Ready” prompt appears.

- . (period) This will perform the same function as “LIST.<ENTER>”, which will instruct BASIC to list the currently active line.
- , (comma) This will perform the same function as “EDIT.<ENTER>”, which will instruct BASIC to enter the “edit mode” for the currently active line.
- <↑> This will cause BASIC to display the next lower numbered line in the program.
- <↓> This will cause BASIC to display the next higher numbered line in the program.
- <←> This will cause BASIC to display the first line of the program.
- <⇒> This will cause BASIC to display the last line of the program.

### Initiating automatic input mode

The AUTO statement initiates automatic input mode with line numbering. Its syntax is:

**Auto [linenum][,increment]**

**Statement**

**linenum** Is the BASIC line number to start input; if omitted, “10” will be used.

**increment** Is the interval between line numbers which will be assigned; if omitted, “10” will be used.

AUTO is the standard way of initiating the input of program lines. This procedure automatically displays the current input line number and awaits your input. If the current line number is a line already in your program, AUTO will display an asterisk, "\*" following the line number. The new line typed will replace the previous program line.

When you complete your entry, the <ENTER> causes BASIC to add your new program line to the existing program, if any, and displays the next line number according to the increment.

When you have completed entering your program lines, press <BREAK> to terminate AUTO and return to the BASIC prompt.

A program line may also be entered without using AUTO by typing a line number followed by the program line.

### Deleting program lines

The DELETE statement is used to remove one or more lines of a program. Its syntax is:

Delete line1[-line2]		Statement
line1	A single line to delete or the first line number of a range of lines.	
line2	The last line number of the range of lines to delete.	

For example, the statement **DELETE 110** will remove the single line numbered 110. **DELETE 120-200** will remove all lines numbered 110 through the line numbered 200

### Editing existing BASIC program lines

This EDIT interpreter BASIC statement is used to invoke the source program line editor.

**Edit line**

Statement

**line**        Is the number of the line you wish to edit.

The **EDIT** command allows editing of a particular line on a mostly conceptual basis (as opposed to directly perceptual screen editing.) 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**

<b>&lt;space&gt;</b>	Skip over next character, displaying it
<b>&lt;←&gt;</b>	In edit mode: Move cursor left nondestructively
<b>&lt;⇒&gt;</b>	In insert mode: Move cursor left destructively
<b>A</b>	Leave the edit with the old line untouched
<b>C&lt;char&gt;</b>	Change characters
<b>D</b>	Delete character
<b>H</b>	Hack line
<b>I</b>	Go into insert mode
<b>K&lt;char&gt;</b>	Delete up to <char>
<b>L</b>	List rest of line and restart edit on new line
<b>S&lt;char&gt;</b>	Move cursor to occurrence of <char> after cursor
<b>X</b>	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 **<SPACEBAR>**. 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-ŀ>**).

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-ŀ>** 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.

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 six characters, say, type **<6><D>**. To "erase" a number just typed and essentially set it back to one, 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.) And, say, **<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.

### **Deleting a disk file**

**KILL** will delete the designated file from the disk directory. Its syntax is:

**KILL"filespec\$"**

Statement

**filespec\$** Designates the file to remove; it may be represented as a string constant or a string expression.

The **KILL** command will allow you to delete a file from a disk directory, making that file inaccessible, and freeing up the space on the diskette that the file consumed. The **KILL** command functions identically to the DOS commands "KILL" or "REMOVE".

Suppose you wish to remove the file MYFILE/DAT from the diskette currently in drive 1, and free up the space consumed by that file. The following command will perform this function.

**KILL"MYFILE/DAT:1"**

Realize that after the deletion is performed, you will no longer be able to access any information which was previously stored in the file. Also note that since the filespec is being represented as a string constant, it must be enclosed in quotes.

### Listing programs

The "LIST" statement is used to display one or more lines of your source program to the video screen; "LLIST" directs the listing to the line printer. Their syntax is:

<b>LIST [line1 ][-][line2]</b>	<b>Statement</b>
<b>LLIST [line1 ][-][line2]</b>	<b>Statement</b>
<b>line1</b>	This is the first line of a range of lines to list.
<b>line2</b>	This is the last of a range of lines to list.

Specifying "LIST" or "LLIST" with no parameter will list the entire program. You can list a portion of the program using one of the following forms:

<b>line</b>	List a single line
<b>line-</b>	List from line to the end
<b>-line</b>	List from the beginning to line
<b>line1-line2</b>	List from line1 through line2

### **Clearing the resident program**

**NEW** deletes the current program from memory. Its syntax is:

<b>NEW</b>	There is no operand!	Statement
------------	----------------------	-----------

### **Renaming a file**

**NAME** is a DOS 6 interpreter BASIC statement used to rename a disk file. It operates like the DOS **RENAME** command. Its syntax is:

<b>DOS 6 Interpreter BASIC</b>	
<b>NAME oldnam\$ AS newnam\$</b>	Statement 6
<b>oldnam\$</b> Is the current name of the file to be renamed.	
<b>newnam\$</b> Is the new desired name.	

### **Copying, Moving, and Searching**

The following four commands are contained in the **BASIC/OV2** file of DOS version 6. The two edit commands of DOS version 5 BASIC are contained in the **BASIC/OV3** overlay file. This file must be present when using these commands, or a File not found error will occur. Like other BASIC editing commands, the use of these will clear all variable values and close any open files.

The **Copy** command will duplicate a single line. Its syntax is:

<b>C Num1,Num2</b>
--------------------

*Num1* is an existing line number to be copied. *Num2* is the line number to create, and must not already exist. No renumbering will be done after the copy. If the line numbers are incorrect, an illegal function call error will occur.



The DOS version 6 BASIC Find command finds all references to a line, variable or keyword. Its syntax is:

### DOS 6 Interpreter BASIC

#### F Object

**Object**     Is a line number, variable name, or keyword.

*Object* is either a line number, variable, or keyword. The space after the "F" is mandatory when finding keywords. The resulting display will be all line numbers containing the referenced object. When finding variables, only the first 10 characters of the variable name will be significant. Also, type declarations !,%,#,\$,( must be used. For example, the command "F A" would not find A\$ or A().

The Move command moves a single line (DOS version 5 BASIC) or a block of lines (DOS version 6 BASIC). Its syntax is:

### Interpreter BASIC

**M Num1,Num2,Num3**     6

**M Num1,Num3**     5

**Num1**     Is the number of the first line to move.

**Num2**     Is the number of the last line in a block of lines.

**Num3**     Is the line preceding the new block location or the new line in the case of a DOS 5 BASIC.

Under DOS 6 BASIC, *Num1* and *Num2* are existing line numbers and define the block of lines to be moved. *Num2* must be greater or equal to *Num1*. *Num3* is an existing line number and is the line to insert the moved block after. The moved block of lines will be renumbered by one, and all references to these lines (if any) will be corrected. If there is not enough room in memory to move the lines, an Out of memory error will occur. If this happens, do multiple moves of smaller pieces. If the line numbers are non-existent or if there is not enough room between *Num3* and its

following line to fit the block, an illegal function call error will occur. For example, if you had program lines 100 and 110, a block of lines moved after line 100 could be no more than 9 lines long.

Under DOS version 5 BASIC, line *Num1* is moved to be line *Num3*. no automatic renumbering of imbedded line number targets is done so you will still have to adjust the targets of any GOTOs, GOSUBs, etc.

The DOS version 6 BASIC **Search** command will search the program and display a reference to a line number, variable name, or keyword. Its syntax is:

### DOS 6 Interpreter BASIC

**S [Object]**

**Object**     Is a line number, variable name, or keyword.

*Object* is a line number, variable or keyword. The first line containing the object will be displayed. The space after the **S** is mandatory when searching for keywords. The **S** with no object following will search for the next occurrence of the previous object. Like the "Find" command, variables are limited to 10 significant characters, and any explicit type declarations must be used.

### BASIC Program Renumbering

This DOS interpreter BASIC feature will renumber BASIC program line numbers as well as correctly adjusting all line number references such as GOSUB and GOTO. The syntax is:

Interpreter BASIC	
<b>CMD"N ! line,newline,inc,last"</b>	Statement 5
<b>RENUM [newline][[,line] [,inc] [,last]]</b>	Statement 6
<b>!</b>	Optional V5 parameter to skip the complete scan for errors before renumbering begins.
<b>newline</b>	Is the new starting line number of the program; if omitted, the default is line 10 for V6 and 20 for V5.
<b>line</b>	Is the starting line number of the program lines to renumber; if omitted, the entire program is renumbered, i.e. line=1, and last = 65529.
<b>inc</b>	Is the line number increment between lines; if omitted, an increment of 10 for V6 and 20 for V5 is used.
<b>last</b>	Is the last line of the program to be renumbered; if omitted, renumbering occurs to the end.

When you have two existing program lines whose line numbers differ by one, and you want to insert another line between them, you must change the existing line numbers. The RENUM or CMD"N..." statement allows for renumbering program line numbers, which is a useful operation during program development. This feature will allow you to renumber all or parts of the BASIC program currently in memory. The lines to be renumbered can be anywhere in the program. If used, all references in GOTO's, GOSUB's, etc., will be properly adjusted. However, if the parameters you use would result in the renumbered lines being out of sequence, a Bad parameters or illegal function call error will occur.

For DOS version 5 BASIC, you cannot have a line number zero (0) if renumbering a program. Also, Both BASIC/CMD and BASIC/OV1 must be present on the disk, or a "Program Not Found" error will occur. If you do not specify the exclamation point, "!", character, a full scan for errors will be done before the renumbering starts. If errors do exist, no lines will be changed. It is usually much easier to fix the errors before the lines are renumbered! If you do specify the "!", any error found will still abort the renumbering. However, all internal line number references will have already been changed up to the line that cause the error. Do not use the "!" parameter unless you are absolutely sure that no errors exist.

For example, the statement:

<b>RENUM 1000,500,10,600</b>	<b>V6</b>
<b>CMD"N 500,1000,10,600</b>	<b>V5</b>

will renumber the lines between 500 and 600 only, making new line numbers starting at 1000 with an increment of 10.

### BASIC Cross Reference Utility

The cross reference facility will allow you to produce a list of the variable and line number references of an BASIC program. DOS version 5 provides this facility as BASIC's "CMD"X"" statement; BREF/CMD is the DOS version 6 BASIC cross reference utility which is executed at the DOS Ready prompt, not from inside of BASIC. The BASIC program must **not** have been saved in ASCII.

The DOS 6 syntax is:

DOS 6 Interpreter BASIC	
<b>BREF filespec ((Var=sw,Line=sw,P=sw,W=n))</b>	
<b>Var=</b>	Cross references variables; the default is ON.
<b>Line=</b>	Cross references line numbers; the default is OFF.
<b>P=ON</b>	Directs the output to a printer; the default is OFF.
<b>W=n</b>	Sets a width for printer output; the default is 80.

The *filespec* is the name of the BASIC program. The **VAR** parameter allows variables to be cross referenced. The **LINE** parameter includes a cross reference of line numbers. The **P** parameter allows the listing to go to a printer rather than the video. The **W** parameter can be used to specify the number of columns for the printer (generally either 80 or 132, although any values between 32 and 255 will be accepted).

Variable names will be displayed up to 14 characters. If the variable is longer than this, the remaining characters will be truncated for display. BASIC stores certain keywords in ASCII, and this makes them indistinguishable from a variable. For example, the "AS" used in field statements is stored in ASCII rather than a token. This will cause these words to be displayed in a variable cross reference list.

If there are more references than can be displayed on a single line, they will wrap to the next line. This line will have an asterisk in column one to denote the overflow.

The command, **BREF PROG/BAS**, sends only variable references to the video. The command, **BREF PROG/BAS (LINE,VAR=OFF)**, sends only line number references to the video. **BREF PROG/BAS (LINE,P,W=132)** Sends variables and line numbers to a 132 column printer.

If you try to use a program saved in ASCII, a program saved in the protected mode, or a non-BASIC program file, the error message, Not a BASIC program, will be generated by BREF.

If the program is too large to fit into your available memory, the error message, Out of memory – can't cross reference, will be generated by BREF. The cure is to change your configuration to free up some memory.

The message, Line nnnn, Error in original program, may occur if there is a syntax error in the original BASIC program. The line number should correspond to that line in the program.

The syntax of the DOS 5 BASIC CMD"X" feature is:

### DOS 5 Interpreter BASIC

**CMD"X devspec/filespec -V | =var, -L | =lnum, <title>"**

**devspec** Is the device or file the listing will be sent to.

**filespec** If not specified, it will go to the video screen.

**-V** All Variables.

**=var** Only the variable specified.

**-L** All Line numbers.

**=lnum** Only the line number specified.

**<title>** Is a title to be printed on the top of each page.

Both BASIC/CMD and BASIC/OV2 must be present on a disk or a "Program Not Found" error will occur. You cannot have a line number zero (0) if you wish to use the cross reference utility.

This list may be sent to any device in the system, such as the \*DO (video screen), \*PR (line printer), etc. It may also be sent directly to a specified disk file. If sent to a file, CMD"X" will use the default extension of /TXT.

Parameters are allowed to determine which variables or line numbers will be listed. If no parameter is specified, all variables and line numbers will be cross referenced.

If you wish a title to be put on the top of every page in the list, it must be specified between less-than/greater-than symbols in the command line.

### **Listing program variables**

The DOS version 5 CMD"V" BASIC extension can be used to dump a list of active variables and their values and user defined functions while a program is running (or after it was interrupted or ended). The syntax of this command is:

#### **DOS 5 Interpreter BASIC**

**CMD"V [\*DO | \*PR] [-S] [-A] [=x]"**

<b>*DO   *PR</b>	Designates the output to either device [*DO]
<b>-S</b>	Restricts output to scalars only.
<b>-A</b>	Restricts output to array variables only
<b>=x</b>	Restricts output to variables starting with "x".

## **BASIC Compiler: Editing and Compiling**

### **CED General Information**

The Compiler BASIC editor differs somewhat from the interpreter BASIC editor. However, all internal editing commands (with the "E" command) are the same. The significant difference between the interpreter BASIC editor and the Compiler BASIC 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 Compiler BASIC editor. Every line is numbered from 1 through "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. This carries the advantage of never having to renumber due to line numbers too close together; the disadvantage lies in the fact that "renumbering" occurs automatically whenever you insert, delete, copy, or move lines - so 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) - the same as a "NEW".

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" in lieu 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.

Entering a NULL string for CED's Find and Search commands results in finding/searching the next occurrence of the previous find/search string, if any.

**GO**

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

<b>Edit &lt;'string'&gt; &lt;linrange&gt;</b>	for BASIC line #'s
<b>ED &lt;'string'&gt; &lt;linrange&gt;</b>	for editor line #'s

The "EDIT" (or abbreviated as "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 will recognize the format of the command, since it is essentially the same as the line EDIT function of the BASIC language on most 8-bit 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**

<b>&lt;space&gt;</b>	Skip over next character, displaying it
<b>&lt;⇐&gt;</b>	In edit mode: Move cursor left nondestructively
<b>&lt;⇐&gt;</b>	In insert mode: Move cursor left destructively
<b>A</b>	Leave the edit with the old line untouched
<b>C&lt;char&gt;</b>	Change characters
<b>D</b>	Delete character
<b>H</b>	Hack line
<b>I</b>	Go into insert mode
<b>K&lt;char&gt;</b>	Delete up to <char>
<b>L</b>	List rest of line and restart edit on new line
<b>S&lt;char&gt;</b>	Move cursor to occurrence of <char> after cursor
<b>X</b>	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-⏏>**).

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-⏏>** 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.

With Compiler BASIC, 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'['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 six characters, say, type <6><D>. To "erase" a number just typed and essentially set it back to one, 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.) And, say, <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.

**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. If "filespec" is omitted, the current file is loaded. The "current" file is noted by "?F". 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.

CED accepts either a pure ASCII file or its own tokenized format. Since BC can be used without regard to the CED editor, it also accepts either pure ASCII files or CED's tokenized files as its source stream input. Please don't expect to edit a BASIC program for use by BC with the interpretive BASIC editor. The tokenization of interpretive BASIC will create problems.

**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.

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

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

**N [line[,last[,newline[,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.

**NEW**

This command effectively does a delete of text from top to bottom clearing out the entire text buffer.

**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” is 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.

**S[string]**

This command operates the same as "Fstring" except the search starts at the beginning of the text instead of line+1. Entering a NULL string for CED's Find and Search commands results in finding/searching the next occurrence of the previous find/search string, if any.

**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. The "V" command permits viewing the "current" file if "filespec" is omitted. Thus, **V<ENTER>** displays the current ASCII file, whereas, **V#<ENTER>** displays the current compressed file. The "current" file is noted by "?F".

**W:[#]filespec [,line1[,line2]]**

This command writes text from memory to the specified disk file in compressed (tokenized) format. 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. If "filespec" is omitted, the current file is written. The "current" file is noted by "?F".

You can also use the syntax, **W:#filespec<,line1<,line2>>**, to write an ASCII file (i.e one not tokenized).

**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).

### Invoking the REF/CMD utility

The Compiler BASIC 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:

## Compiler BASIC

**REF filespec[/DAT][,-V-L]**

**filespec**     Is the reference data filespec.

**-V**             Directs the REF output to the video screen.

**-L**             Generates 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      AS : 5F99H      A1@ : 6055H      A2@ : 604DH
A3@ : 6045H      AP@ : 5FD5H      B@ (1@) : 5FA9H  BP@ : 5FDDH
C@ (1@) : 5FA1H  CN@ : 60B9H      CP@ : 5FE5H      CS@ : 60C1H
HS : 5F9DH      II : 60A1H      J! : 6029H      KI : 600DH
LI : 60A5H      NI : 6005H      NI! : 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	@@C0 - 7DFAH	@@CF - 87DFH
@@CLNUM - 65E4H	@@CP - 87CDH	@@CT - 87CCH
@@CURBUF - 65DEH	@@DG - 87E1H	
@@DIGBUF - 7E06H	@@DIGPNT - 7E02H	
@@DPFNT - 7E04H	@@DRWRITE - 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		
@@STRCMP3 - 6E30H	@@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	
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			

## **Compilation from CED Editor**

The easiest way to compile a source program is to use CED to create a Compiler BASIC 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 interpreter BASIC editor (i.e., typing all lines verbatim), enter the following (yes, Compiler BASIC supports block graphics on a Model 4).

```
10  '
20  ' Draws design on the screen
30  '
40  CLS
50  FOR Y=0 TO 47 STEP 3
60  '
70  ' Plot lines moving in opposite
   directions from opposing
80  ' corners
90  PLOT S,0,0 TO 127,Y:PLOT S,127,47 TO
   0,47-Y
100 NEXT
110 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 Compiler BASIC 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. Compiler BASIC 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, link, executed, etc. 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 back 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 which is to be invoked from DOS Ready, re-compile the source program using BC/CMD.

### 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 #xxxx

will appear ("xxxx" 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 later in this manual. Note that certain special DOS error codes, different than standard or unique codes, will be flagged by being in the range 32-100, with 32 added to the original code to produce the Compiler BASIC code. The DOS error code must be between 0 and 68 to avoid confusion with other Compiler BASIC error codes.

### Command-line compiling

The general format of a direct compiler invocation is:

Compiler BASIC	
<b>BC filespec,saddr,taddr,-dir-dir,...</b>	
<b>filespec</b>	Is the source program specification. The extension defaults to "/BAS".
<b>saddr</b>	Is the specified program origin (start address).
<b>taddr</b>	Is the top or highest address to be used by the compiled program.
<b>-dir</b>	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 (DOS version 5) or 2600H (DOS version 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 "taddr" 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 over-

writing 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 still retain their default values. However, the commas are necessary to delimit the sentence. "C TEST/BAS -NO" is invalid, as is "C TEST/BAS, -NO" and "C 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 *"\*"*.

### **Compiler-generated line numbers**

The compiler automatically generates line numbers in the executable program generated by the compiler for any source line which has no BASIC line number. These numbers start from "1" and are incremented by "1". This helps identify which line was suspect during a runtime error trap. These line numbers are only used for reporting purposes. Runtime error reporting is not as useful without a reported line number. Your program cannot reference these "pseudo" line numbers. Nothing prohibits your program from assigning a BASIC line number the same as these automatic line numbers; however, for maximum usefulness in error reporting, you should avoid that practice.

### **Compiler Directives**

BC supports the following compiler directives: GET, INCLUDE, IF-ENDIF, INJECT, LINK, LIST, NOLIST, NO, NOPRT, NS, NX, PRINT,

PRT, WD, WE, YS, YX, Z80. In the following paragraphs, directives which are considered global in nature will be denoted with "(G)"; directives which are considered local in nature will be denoted with "(L)"; directives which are considered both local and global with "(B)"; and directives which 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.

### \*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 back to original source program.\*INCLUDE and \*GET compiler directive. Note that the filespec must include any extension, as required.

### 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.

The "EQU" operator of "label" also allows you to define values for labels to be used typically in conditional compilation. For example:

```
"DOS5" EQU 0:"DOS6" EQU 1
*IF DOS5
PRINT"DOS 5"
*ENDIF
*IF DOS6
PRINT "DOS 6"
*ENDIF
```

allows selective compilation of source program lines based on the value of the label tested by the \*IF assembler directive.

### \*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 Compiler BASIC. The SUPPORT/DAT library file is an example of such a link file. In addition to great disk space efficiency, link files are "assembled" much faster than the original source.

### LIST (B)

This directive will list the source program on the video screen 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.

### 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.

### NS (B)

This directive tells the compiler to inhibit the generation of source line number information in the compiled program's object code file. This saves three 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.

### 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.

### \*PRINT[#n] ["info"] [L] [:] [\$ (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.

### 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.

### WD (B)

This directs Compiler BASIC 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 file's generation. This file can be subsequently processed by the REF/CMD utility to produce a program reference report.

### 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 of the video screen. Any keystroke will cause a continuation of the compilation.

### YS (L)

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

### YX (L)

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

### 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.



### **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. It must not be invoked from DOS Ready.

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 number 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.

### **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 publisher, 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.



### BASIC Statements and Functions

This section of the *Reference Manual* contains information on all statements and functions - other than those considered to be editing or programming aids - supported by the various BASIC interpreters and compilers covered by the manual. This section lists each statement and function strictly in alphabetical order so that you may more easily locate the material covering a specific operation.

Since not every statement or function, or a sub-operation of a statement or function, is supported by all versions of BASIC, this section will clarify the specific level of support. To begin with, each *command* is summarized in a command box which looks like the following box.

Supported BASIC		
command and parameters	Type	x
explanation of parameters		

The top line lists the BASICs supporting the command. If some BASICs have restrictions on the command, the specific syntax supported is specified by the codes in the "x" field. These codes are: I=Interpreter BASIC (both DOS 5 and DOS 6), I5=DOS 5 Interpreter BASIC, I6=DOS 6 Interpreter BASIC, C=Compiler BASIC (both DOS 5 and DOS 6), C5=DOS 5 Compiler BASIC, C6=DOS 6 Compiler BASIC. If there is no code in the "x" field, the statement or function is supported in its entirety by the BASICs identified in "Supported BASIC". The *type* of command will be either "Statement" or "Function".

As is common throughout this manual, optional entries are enclosed in square brackets, "[ ]"; one of two permissible selections are shown separated with a vertical bar, "this|that".

# ABS

# ABS

This function returns the absolute value of its argument.

Compiler BASIC and Interpreter BASIC	
<b>ABS(exp)</b>	Function
<b>exp</b>	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:  $\text{ABS}(-4) = 4$ ;  $\text{ABS}(0) = 0$ ;  $\text{ABS}(1.414) = 1.414$ .

# ADDRA

# ADDRA

This Compiler BASIC function obtains the absolute memory address of its argument.

Compiler BASIC	
<b>ADDRA(addr)</b>	Function
<b>addr</b>	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)`.

# ALLOCATE

# ALLOCATE

This Compiler BASIC statement is used to allocate the quantity of disk file control blocks during run-time. Its syntax is:

Compiler BASIC	
ALLOCATE exp	Statement
<b>exp</b>	<i>exp</i> is the number of file control blocks to allocate in the range <1-15>.

Before any disk files can be OPENed under Compiler BASIC, 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 three 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 need 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.

# ASC

# ASC

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

Compiler BASIC and Interpreter BASIC	
<b>ASC(exp\$)</b>	Function
<b>exp\$</b>	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\$.

# ATN

# ATN

This function obtains the trigonometric arc-tangent of its argument.

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>ATN(exp)</b>	<b>Function</b>
<b>exp</b>	Is a numeric expression in radian measure

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



**&B****&B**

This Compiler BASIC function indicate that the argument is a binary number rather than a decimal format number.

Compiler BASIC	
<b>&amp;Bd0...d15 Binary number</b>	<b>Function</b>

**&B** signals a binary number in ASCII format. For example, the assignments:

**A = B AND &B11110101**

and

**A = B AND 245**

are functionally equivalent.

# **BIN\$**

# **BIN\$**

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

<b>Compiler BASIC</b>	
<b>BIN\$(exp16)</b>	<b>Function</b>
<b>exp16</b>	Is in the range <-32768 to 32767>

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

## BKOFF, BKON

## BKOFF, BKON

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

Compiler BASIC	
<b>BKON</b>	Statement
<b>BKOFF</b>	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...."
```

# CALL

# CALL

This command, similar to “USRn”, is used to transfer control to a machine language program; more than one parameter may be passed.

## DOS 6 Interpreter BASIC

**CALL** *addr* [(*parmlist*)]

Statement

**addr** Is the entry point of the machine-language routine; *addr* cannot be an array variable.

**parmlist** Is a list of comma-separated variables whose values are passed to the routine.

CALL is akin to “USRn”. But where a “USR” can pass only a single variable to a machine-language routine, “CALL” can pass any number of variables. The quantity and type of the variables expected by the routine must be matched by the calling program. The routine transfers control back to the BASIC program with a Z80 “RET” instruction.

If from one to three parameters are passed in the *parmlist*, pointers to the variables are passed in registers HL, DE, and BC for parameter 1, 2, and 3 respectively. Note that these are pointers to where the variables are stored (see argument storage in “USR” and “VARPTR”).

For a *parmlist* of more than three parameters, registers HL and DE contain pointers to parameter 1 and 2; register BC contains a pointer to a data block containing the word addresses of pointers to parameters three through the last parameter identified in the *parmlist*.

In the following example, a call is made to a subroutine at address XF123; three variables are passed: a single precision, an integer, and a string.

```
1000 ENTRYPT = &HF123
1010 CALL ENTRYPT (VAR$NG!, VARINT%, VARSTR$)
...
```

## **CDBL**

## **CDBL**

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

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>CDBL(exp)</b>	<b>Function</b>
<b>exp</b>	Is a numeric expression.

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

## CHAIN

## CHAIN

This DOS 6 Interpreter BASIC statement is used to automatically load and/or merge a BASIC program while passing current variables to it.

### DOS 6 Interpreter BASIC

**CHAIN [MERGE] "filespec" [,line][,ALL][,DELETE line1-line2]**  
Statement

<b>MERGE</b>	Specifies that lines of "filespec" will overlay like-numbered lines of the main program (see MERGE statement for further details).
<b>filespec</b>	Is the BASIC program, saved in ASCII format, which is loaded.
<b>line</b>	Is the line number of "filespec" to begin running; if omitted, execution begins with the first line.
<b>ALL</b>	Designates that all variables in the main program will be passed to the chained program. Without use of <i>ALL</i> , variables can be passed with "COMMON".
<b>DELETE</b>	Used usually with <i>MERGE</i> ; specifies that <i>line1</i> through <i>line2</i> will be deleted from the current program before the chained program is loaded.

**CHAIN** can be used by the programmer to overcome the limitation of limited memory. By partitioning a program into separate modules, each module can be selectively run by a main module; some or all of the "current" program's variables can be passed to subsequent modules.

The statement, **CHAIN "PROG2/BAS"** is equivalent to **RUN "PROG2/BAS"**; however, the statement, **CHAIN "PROG2/BAS",1000** has no direct "RUN" equivalent. Equivalence could be achieved by having **GOTO 1000** as the first statement of PROG2/BAS. Thus, the "CHAIN"

statement allows you to control the statement of the chained program which will be first executed. See "COMMON" for an example.

When the "MERGE" parameter is used with "CHAIN", the current program is not erased from memory; lines of the chained program overlay like-numbered lines of the program currently in memory - the program issuing the "CHAIN" statement. This behavior is exactly like that of the "MERGE" statement used alone. However, "CHAIN" adds the flexibility of being able to pass the values of variables from the current program to the chained program by using either the "ALL" parameter, in which all variables will be passed, or by using identical "COMMON" statements in each sub-program being chained. For instance, the statement,

**CHAIN MERGE "PROG2/BAS",100,ALL**

will merge the ASCII-saved program, PROG2/BAS, into the current one, pass the values of all variables to the newly-configured program in memory, and begin execution at line 100.

If you wish to erase a block of lines from the current program prior to loading the chained program, use the "DELETE" parameter. Conceptually, it is wise to consider using a short main program with line numbers in the range 10-999, and use sub-program modules with line numbers 1000 and above. Thus, you could completely remove all traces of the current sub-program under control of the main module with a statement such as:

**CHAIN MERGE "SUB4/BAS",1000,ALL,DELETE 1000-60000**

This example will delete all lines numbered 1000 through 60000 from the current program, merge in SUB4/BAS, pass all variables to the new configuration, then begin execution at line 1000.

When you specify the "MERGE" parameter, existing defined functions (i.e. with DEF FN) will be retained. This requires that functions defined by "DEF FN" be positioned prior to the "CHAIN MERGE" statement. Also, variables declared by "DEFINT", "DEFSNG", "DEFDBL", and "DEFSTR" will be retained. Using "MERGE" will also keep the current "OPTION BASE" setting (for array base values); existing OPEN files will be left OPEN. None of these actions occur if you omit the "MERGE" parameter.

## CHR\$

## CHR\$

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

Compiler BASIC and Interpreter BASIC	
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.



# CINT

# CINT

This function converts a numeric expression to integer format.

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>CINT(exp)</b>	<b>Function</b>
<b>exp</b>	<b>Is a numeric expression.</b>

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

# CLEAR

# CLEAR

The **CLEAR** statement is used to clear variables and allocate string and/or stack space.

## Compiler BASIC and Interpreter BASIC

**CLEAR [exp]** Statement C,5

**CLEAR [,mem][,stack]** Statement 6

**mem** Sets the highest memory address to be used by BASIC. All memory above this address will be "protected". If not specified, all memory up to **HIGH\$** will be available.

**stack** Establishes the amount of space used by BASIC for internal use handling FOR-NEXT loops, calls, etc; if not specified, the default is 512 or 1/8th the available memory.

**exp** Designates 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** performs differently depending on the implementation. For Compiler BASIC and DOS 5 Interpreter BASIC, "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 eight 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 "scratch pad", so a buffer of 600 bytes is not unreasonable -- make it: **CLEAR 4600**.

Under DOS 6 Interpreter BASIC, **CLEAR** uses parameters of "mem" and "stack". The "mem" parameter establishes the highest address used by

BASIC. This is identical to the "H" parameter optionally passed when invoking BASIC.

The "stack" parameter fixes the amount of internal stack space to be made available from the free memory pool (i.e. that obtained initially from ?MEM+512). If the stack parameter is not specified, BASIC will establish a stack space of either 512 bytes or one-eighth the available memory, whichever is the smaller amount.

In the case of either DOS 5 or 6, since "CLEAR" will zero all variables and close all files, it should only be done at the beginning of your program before any variables have been defined.

# CLOSE

# CLOSE

This statement is used to close a file or files. The syntax for the CLOSE statement is as follows:

## Compiler BASIC and Interpreter BASIC

**CLOSE** [*bufnum* [,*bufnum* ...]]

Statement

**bufnum**     Designates a specific file to close. If no *bufnum* 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.)

The "CLOSE" command is used in conjunction with the "OPEN" command. After a file has been opened, it is capable of being read from and/or written to. To disable this read/write capability of a disk file, a "CLOSE" of the file must be done. In addition, "CLOSE" will update the Mod flag, Mod date and end of file marker in the directory record of that file (provided the file has been written to). See "OPEN" for more information on file access.

With a list of file buffer numbers given, such as "CLOSE #,...,#", only those files that have been opened with the specified buffer number (where "#" represents the buffer number used to define a particular file in an OPEN statement). Closed file buffers are unaffected by "CLOSE". With no specific file buffer numbers listed, all open files are closed.

If you issue any command which will perform a "CLEAR" (such as "EDIT", "CLEAR", or "RUN"), a global "CLOSE" will automatically be performed for you. However, if you issue a CMD"S", CMD"A", or CMD"I" command, closing of any open files will not occur. For this reason, you should always make sure files have been closed prior to exiting back to the DOS Ready prompt.

# CLS

# CLS

This statement is used to clear the video display screen.

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>CLS</b>	<b>Statement</b>

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.

## CMD

## CMD

The DOS 5 Interpreter BASIC CMD command allows you to perform certain DOS library and utility commands without having to leave BASIC. In addition, there are many distinct parameters that may be used in conjunction with the "CMD" command which will allow you to perform various different functions. To invoke a DOS library command from DOS 6 Interpreter BASIC, see "SYSTEM"; Compiler BASIC also uses "SYSTEM". The syntax used for the "CMD" command is as follows:

## DOS 5 Interpreter BASIC

CMD"command"

Statement

CMD"x"

Statement

**command** Is a DOS command to invoke.**x** Is the letter assigned to the special command.

## CMD"command"

DOS library commands and utilities that do not affect HIGH\$ may be executed from BASIC by use of the CMD"command". The following examples should illustrate implementation of this feature

CMD"DIR :0"

Displays a directory of drive :0.

CMD"DEVICE"

Will display the device table.

CMD"LIST DAT1/SCR"

Will list the file DAT1/SCR.

CMD"BACKUP :0 :1"

Will perform the designated Backup.

After the desired DOS function has been completed, control will be returned to BASIC with your program and variables intact. This type of CMD command will function whether it is called from BASIC's command line or from within a BASIC program. If performed from within an BASIC program and an error occurs, or the CMD command is aborted with the <BREAK> key prior to being completed, the appropriate error message will be displayed, or the message "System Command Aborted" will appear, and execution of the BASIC program in question will be terminated. The

command may also be contained within a string variable, such as the following format:

**A\$="DIR :0":CMD A\$**

Approximately 4K of free memory must be available for these types of CMD commands, or an "Out of Memory" error will occur.

**CMD\*\*\***

This command will send the contents of the screen to the printer. This will allow you to perform a screen print from within a program, without having to physically initiate the screen print. CMD\*\*\* may also be issued from the BASIC Ready prompt. Note that the JKL parameter of the KI/DVR need not be active to utilize this command.

**CMD"A"**

This command will perform an abnormal return to DOS. Any active DO command will be cancelled.

**CMD"B","switch"**

This command will enable or disable the <BREAK> key, with "switch" being either "ON" or "OFF". A string constant or string expression may be used to represent the "switch".

**CMD"D"**

Turns on and enters the system debugger.

**CMD"D","switch"**

This command is similar to the CMD"D" command, with the following exceptions. The switch "ON" will turn on the system debugger, but will remain in BASIC. Pressing the <BREAK> key (or <CLEAR> <SHIFT> <D> keys if MiniDOS is active) will cause you to enter the debugger. The switch "OFF" will turn off the debugger.

### **CMD"E"**

This command will return the last DOS error message encountered. If no error has been encountered, the message "No Error" will appear. CMD"E" may be useful when you wish to pinpoint the exact nature of an error. BASIC's error dictionary is not as extensive as that found in DOS, hence various DOS errors can produce the same BASIC error message. Performing a CMD"E" will give you the exact error seen by DOS. This may be of use when you get the BASIC error message "Disk full or write protected" or "Disk I/O error".

### **CMD"I","command"**

This command functions much the same as the CMD"command", with the exception that control will return to DOS after the *command* has been executed. *Command* can be represented as a string constant or a string expression. If represented as a string constant, it must be contained within quotes.

### **CMD"L","filespec"**

This command will load a Load Module Format file (a machine language program) into memory, much the same as the DOS LOAD command does. *Filespec* may be represented as a string constant or a string expression. If represented as a string constant, it must be contained within quotes.

### **CMD"N"**

This command provides you with a program line renumbering function. For the specific parameters involved with this command, please refer to the section on *Editing Interpreter BASIC*.

### **CMD"O",numelem, firstelem**

This command will allow you to sort a singly-dimensioned string array. The sort will start at the element specified, and will sort the number of elements specified. The number of elements to be sorted must not force the sort past the end of the array. In order to utilize the CMD"O" function, the BASIC/OV3 module must be present on a disk in the system.



```
CMD"O", 15, A$(10)  
X=15:Y=10:CMD"O", X, A$(Y)
```

Issuing either of the above commands will cause a sort to be performed on the A\$ array. After the sort has been finished, elements 10-24 will be sorted in alphabetical order.

### **CMD"P",variable**

This command will return the **printer status** in the variable specified. The variable may be any type, including a string. The value will have the bottom four bits stripped before being passed back to BASIC.

### **CMD"R"**

Model I - This command enables the interrupts. It should be performed after a CMD"T" command has been issued. For more information see the CMD"T" command.

Model III - This command will turn on the clock display.

### **CMD"S"**

This command is the normal way to return to DOS Ready from BASIC.

### **CMD"T"**

Model I - This command will disable the interrupts. It must be issued prior to performing cassette tape I/O. After the tape I/O has been completed, the interrupts must be enabled with the CMD"R" command.

Model III - This command will turn off the clock display.

### **CMD"X"**

This command provides you with a program cross reference facility. For the specific parameters involved with this command, please refer to the section on *Editing Interpreter BASIC*.

# COMMAND

# COMMAND

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

## Compiler BASIC

<b>COMMAND name(varlist)</b>	Statement
<b>program statements</b>	
<b>ENDCOM</b>	Statement

**name**      Is a string of characters in the set ("A"-“Z”, “0”-“9”), starting with (“A”-“Z”)

**varlist**      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 or 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 : RET
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 ...')

## COMMON

## COMMON

COMMON is a DOS 6 Interpreter BASIC statement used in conjunction with the "CHAIN" statement to pass selected variables to a chained program. Its syntax is:

DOS 6 Interpreter BASIC	
<b>COMMON var[,var,...]</b>	Statement
<b>var</b>	Is a scalar variable or array to be passed to a chained program.

The "CHAIN" statement permits you pass all variables to a chained program by means of the "ALL" parameter. When you wish to pass a selected set of variables, use an identical COMMON statement in each of the programs being chained and don't use "ALL" in the "CHAIN" statement.

Scalar variables are identified by means of the variable name; arrays are indicated by using the array name without a subscript. For example,

**COMMON A\$,B\$,C%()**

passes the two string variables, A\$ and B\$, and the integer array, C%(). Note the use of this statement in the following example:

```
10 REM This is program ONE/BAS
20 COMMON A$,B$,C%()
30 B$="ONE/BAS"
40 A$="program two"
50 C%(2)=C%(2)+100
60 PRINT "In program one with C%(2) = ";C%(2)
70 CHAIN "TWO/BAS"
```

Program ONE/BAS utilizes a "COMMON" statement to pass two scalar variables, A\$ and B\$, and one array variable, C%(). This nonsense program, designed solely to illustrate the use of "COMMON" with chained programs, simply creates assignments for the two string variables and proceeds to increment one of the array elements. The program then

chains to the second program, TWO/BAS. Since ONE/BAS is also a program chained from another, ONE/BAS needs to be saved in ASCII format with a command such as, **SAVE"ONE/BAS",A**.

```
10 REM This is program TWO/BAS
20 COMMON A$,B$,C%( )
30 PRINT "In ";B$;" with C%(2) = ";C%(2)
40 CHAIN B$,40
```

Program TWO/BAS utilizes a COMMON statement identical to ONE/BAS; this enables it to retain the identified variables in common with the chaining program. TWO/BAS simply prints the current value of the array element incremented in ONE/BAS; its PRINT statement utilizes the string variable assigned in ONE/BAS. TWO/BAS then chains back to ONE/BAS to begin execution at line numbered 40. Note that TWO/BAS also must be saved in ASCII using **SAVE"TWO/BAS",A**.

When ONE/BAS is RUN, execution will swap back and forth between ONE/BAS and TWO/BAS until you press the <BREAK> key or the value of C%(2) overflows an integer value resulting in an error.

# COMPL

# COMPL

This Compiler BASIC statement is used to complement a pixel.

Compiler BASIC	
COMPL(x,y)	Statement
<b>x</b>	Is a numeric expression which evaluates to the range <0 - 127> for 64-column screens and <0 - 159> for 80-column screens.
<b>y</b>	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 program illustrates a brief example of these graphics commands:

```
5      Y=23:RANDOM:CLS
10     FOR X=0 TO 127: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:COMPL(X,23):NEXT
80     FOR X=0 TO 127: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.

# CONT

# CONT

This Interpreter BASIC statement is used to continue a program after a <BREAK>, "STOP", or "END".

Interpreter BASIC	
CONT	Statement

The CONT statement, usually useful only during program development, allows you to continue execution of a program after it is prematurely stopped by means of a <BREAK> key operation, programmed "STOP" statement, or program "END" statement. Prior to the "CONT", you may print the contents of variables or change the value of variables by means of direct mode assignments. You may **not** edit any program line if you wish to continue the execution of your program.



# **COS**

# **COS**

This function obtains the trigonometric cosine of its argument.

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>COS(exp)</b>	<b>Function</b>
<b>exp</b>	Is a numeric expression in radian measure.

**COS** takes the cosine, in radians, of an expression. Interpreter BASIC will return a single precision result. Compiler BASIC will return, 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.

## CSNG

## CSNG

This function converts its argument to single precision.

Compiler BASIC and Interpreter BASIC	
CSNG(exp)	Function
exp	Is a numeric expression.

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

# CURLOC

# CURLOC

This function obtains the position cursor of the video cursor.

Compiler BASIC		
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 value returned by "CURLOC" would be equivalent to the result of  $(ROW(0)*NCOL)+(POS(0)-1)$ , where *NCOL* is the number of video columns, 64 or 80.

## **CVD**

## **CVD**

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

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>CVD(exp8\$)</b>	<b>Function</b>
<b>exp8\$</b>	Is an eight-byte string expression.

CVD's primary purpose is to convert a double precision number stored in a file on disk as an eight-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. As a precise definition,

**var = CVD(MKD\$(var))**

When using "CVD", the eight-byte string should be a representation of a double precision number stored in compressed format, typically retrieved from a disk file (in essence, it performs the opposite function of the "MKD\$" command). For more information on storing a double precision number in compressed format in a disk file, refer to the "MKD\$" command.

For example, assume that "A\$" is an eight-byte string which represents a compressed double precision number. After the command,

**A#=CVD(A\$)**

is performed, "A#" will be set equal to the uncompressed number that "A\$" represents.

Realize that you are not limited in using CVD to assign a value to a variable. The value generated by a CVD command may be used directly (e.g. **PRINT CVD(A\$)**, or **IF CVD(A\$)<100000 THEN GOTO 1000**).

## **CVI**

## **CVI**

The **CVI** function unpacks the two-byte string argument to an integer number.

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>CVI(exp2\$)</b>	<b>Function</b>
<b>exp2\$</b>	Is a two-byte string expression.

The main purpose of **CVI** is to convert an integer stored as a two-byte string on disk by the converse string function **MKI\$(exp)** back into an integer. As a precise definition,

**var = CVI(MKI\$(var))**

The “**CVI**” command functions identically to the “**CVD**” command with the following exceptions. The “**CVI**” command will convert a two-byte string into an integer. This two byte string should be a representation of an integer stored in compressed format. “**CVI**” performs the opposite function of the “**MKI\$**” command. The value returned from the “**CVI**” function will be an integer within the range of -32768 to +32767 inclusive. For more information, refer to the “**MKI\$**” command.

For example, assume that “**A\$**” is a two-byte string which represents a compressed integer. After the command,

**A%=CVI(A\$)**

is performed, “**A%**” will be set equal to the uncompressed number that “**A\$**” represents.

## CVS

## CVS

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

Compiler BASIC and Interpreter BASIC	
<b>CVS(exp4\$)</b>	Function
<b>exp4\$</b>	Is a four-byte string expression.

The prime function of CVS is to convert a single precision number, converted into a four-byte string by the "MKS\$" string function and stored in a disk file, back into a single precision number. To be precise, "CVS" can be defined as,

**var = CVS(MKS\$(var))**

The "CVS" command functions identically to the "CVD" command with the following exceptions. The "CVS" command will convert a four-byte string into a single precision number. This four-byte string should be a representation of a single precision number stored in compressed format. "CVS" performs the opposite function of the "MKS\$" command. For more information, refer to the "MKS\$" command.

For example, assume that "A\$" is a four-byte string which represents a compressed single precision number. After the command,

**AI=CVS(A\$)**

is performed, "AI" will be set equal to the uncompressed number that "A\$" represents.

## DATA

## DATA

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

### Compiler BASIC and Interpreter BASIC

#### 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” (Compiler BASIC) provide ways to point at the desired data list. **RDGOTO *addr*** and **RESTORE *line***, especially, eliminate the wasteful process of reading and discarding lists of data to get to the desired list in BASIC.

Initially, the first data item read, unless the data pointer is changed by a **RDGOTO *addr*** or **RESTORE *line*** statement, will be the first data item in the first “DATA” statement in the program.

### Example Compiler BASIC 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
```

### Example Interpreter BASIC Program:

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



# DATE\$

# DATE\$

This function returns the system date as a string.

Compiler BASIC and DOS 6 Interpreter BASIC		
DATE\$	There is no operand	Function

When using the DATE\$ function, the system date is returned as an eight-character string of the form, "MM/DD/YY". Note that the YY field captures years from 2000 through 2011 as "00" through "11".

For instance if today is Thursday, March 19, 1992,

**PRINT DATE\$**

will display "03/19/92".

# DEC

# DEC

DEC is a Compiler BASIC statement used to decrement an integer variable.

Compiler BASIC	
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
```

## **DEFDBL,    DEFINT,    DEFSNG,    DEFSTR**

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

### **Compiler BASIC and Interpreter BASIC**

<b>DEFDBL letters</b>	<b>Statement</b>
<b>DEFINT letters</b>	<b>Statement</b>
<b>DEFSNG letters</b>	<b>Statement</b>
<b>DEFSTR letters</b>	<b>Statement</b>

**letters**      Is a list of letters (A-Z) flagging all variables beginning with the specified letter(s). 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, is single precision. Type declaration characters are: “%” = integer with two bytes of storage needed; “!” = single precision with four bytes of storage needed; “\$” = string with three or four bytes of storage needed; and “#” = double precision with eight bytes of storage needed). 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 BASIC to assume that all following untyped variables starting with one of the letters “A”, “B”, “C”, “D”, “E”, “F”, “G”, “H”, “I”, “J”, or “K” are integers (integer type).

### **Important Compiler Incompatibility Note:**

DEFXXX statements are compiler pseudo-ops! They affect output as they are **linearly** encountered sequentially in a source line, not as they are **logically** encountered during program execution. In other words, **runtime** logic has absolutely nothing to do with setting untyped variable type defaults, unlike interpretive BASIC.

## DEFFN

## DEFFN

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

### Compiler BASIC and Interpreter BASIC

**DEFFN name(input variable list) = exp**      Statement C,I5

**DEF FN name(input variable list) = exp**      Statement I

**Input variable** Is any simple string or numeric variable. Arrays are not allowed.

Note: DOS 6 Interpreter BASIC requires a SPACE character between "DEF" and "FN"!

There are many intrinsic functions provided for you in BASIC (i.e. "VAL", "STR\$", "SIN", etc.). **DEFFN** or **DEF FN** allows you to define your own functions capable of being evaluated from a single expression. This may be of use when performing lengthy calculations at different points in your program when you do not use the same variable names to perform these calculations.

The function "name" is the name that you will assign to the function, and has the same restrictions as those imposed on a variable name. The function name must be of the same type as the value to be returned from the function.

The "(input variable list)" is a list of variables to be passed to the function. The variable names used are local to the function, and act as dummy variables. They will have no effect on other variables in the program which have the same name. However, they must be of the same variable type as the variable represents in the function (i.e. string, integer, single precision, double precision). Also, if more than one variable is to be passed to the function, they must be passed in the same order as that defined in "(input variable list)" (see example below).

The "exp" is an expression which represents how the variables passed to the function are to be worked on.

This example will show how to create a function which will build a filespec. This function will be passed three variables; the filename, the file extension, and the drive specification. It will return a filespec in the form - "filename/ext:d". A DEF FN statement to create such a function might take on the following format:

```
DEF FNFS$(X$,Y$,Z%)=X$+"/"+Y$+"."+MID$(STR$(Z%),2,1)
```

The function name is "FS\$", and is of string type, since a string value will be returned from the function.

Three values will be passed to the function. The first two values passed will be strings, while the third value will be an integer.

The function that will be performed is as follows: The first string passed to the function will have a "/" added onto the end of it, after which the extension, a ".", and the drivespec will be added to the string, respectively.

The following example will illustrate how to invoke the function, as well as changes that will occur to the variables involved.

```
X$="HELLO" : F$="MYPROG" : E$="BAS" : G%=2  
F1$=FNFS$(F$,E$,G%)  
F2$=FNFS$(E$,F$,G%)
```

After execution of the above three lines, the following variables will be assigned the following values:

```
X$="HELLO"      F$="MYPROG"    E$="BAS"        G%=2  
F1$="MYPROG/BAS:2"  
F2$="BAS/MYPROG:2"
```

Note that the value of "X\$" does not change from the calling of this function. Also note the difference between "F1"\$ and "F2\$". The order in which variables appear when invoking the function determines the value that will be returned from the function.

As a final note on DEF FN, the value returned from the function can be used directly, and does not have to be stored in a variable (e.g. **PRINT FNFS\$(F\$,E\$,G%)** ).

## DEF USR

## DEF USR

This command is used to define the starting address (entry point) of a user created machine language subroutine. A DEFUSR statement must be done prior to utilizing the machine language subroutine via the "USR" command. The syntax for the "DEFUSR" statement is:

### Interpreter BASIC

**DEFUSR[n]=addr**

Statement 15

**DEF USR[n]=addr**

Statement 1

**n** Is a numeric constant (0-9) which is used to identify the machine language subroutine. If omitted, a value of "0" is assumed.

**addr** Is the address which represents the entry point into the machine language subroutine.

**Note:** DOS 6 Interpreter BASIC requires a SPACE character between "DEF" and "USR"!

The number assigned to the subroutine, "n", must be the same as the number used to reference the subroutine with the USR statement.

The entry address to the subroutine may be a constant (i.e. a hexadecimal or decimal number), or it may be a numeric expression. Note that if the starting address is specified as a decimal number, and this address is greater than 32767, it must be specified as the address minus 65536.

Suppose you have a machine language subroutine that has a starting address of &HF000 (61440), and you wish to reference this routine as machine language subroutine number "2". To define this subroutine, one of the following commands may be given:

```
DEF USR2=&HF000
```

```
DEF USR2= (61440-65536)
```

```
DEF USR2= (-4096)
```

# DIM

# DIM

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

## Compiler BASIC and Interpreter BASIC

<b>DIM array(explist) [,array2(explist)...</b>	<b>Statement</b>
--	------------------

<b>array</b>	Is an array name.
--------------	-------------------

<b>explist</b>	Is an expression or list of expressions, specifying the index limits of the array.
----------------	--

Interpretive BASIC defaults to an array size of 10, if an array is referenced without use of a corresponding "DIM".

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)**.

Note that Interpretive BASIC arrays will, by default, utilize a zero base value for the first array element, i.e. array elements are referenced as 0, 1, 2, ... DOS 6 Interpreter BASIC provides the "OPTION BASE" statement to predeclare the array base as either 0 or 1. If you prefer to always use "1" as the first element index, then it is wise to include an "OPTION BASE" statement to reduce array storage requirements.

All Compiler BASIC arrays must be DIMensioned before use, and all DIMs must be in a single "DIM" statement! A compiler trap will abort

compilation if an array reference was found without a corresponding "DIM" statement; the variable name is displayed. The reason why arrays must be DIM'd is that the space they occupy is dynamically allocated by a SUPPORT/DAT subroutine call invoked by the DIM process. There is no such restriction for Interpretive BASIC.

Compiler BASIC 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 Compiler BASIC 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.



# DOWN

# DOWN

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

Compiler BASIC	
DOWN	Statement

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

## DRAW

## DRAW

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

Compiler BASIC	
<b>DRAW flag @ x,y USING array%(exp)</b>	Statement
<b>flag</b>	Designates the type of pixel action: SET signifies unconditional SET; RESET signifies unconditional RESET; COMPL signifies pixel COMPLEMENT.
<b>x,y</b>	Is the coordinate of the starting point. x is in the range (0-127) (0-179); y is in the range (0-4710-71).
<b>array%(exp)</b>	Is an integer array element.

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.

**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 non-integer line lengths and scaling: line lengths are specified in 1/256 block 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 four bytes, encoded into a specified integer array ("**A%**" in this example) in the following manner:

"A%(x) = Byte1 + 256 \* Byte2" where "Byte1" is n/256 fraction of line length and "Byte2" is the integer part of the line length. Byte1 and Byte2 contain the line length information: "BYTE2 + BYTE1/256" is the line length.

"A%(x+1) = Byte3 + 256 \* Byte4" where "Byte3" specifies the rotation number in DRAW degrees (0-255) and "Byte4" is the ENTRY code. Byte3 contains the number of degrees relative to the current orientation to draw the next line. The ENTRY code specified by Byte4 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

```
10 DEFINT F:CLS:DIM FIGURE1(110)
20 Y=0:FOR X=0 TO 250 STEP 10
40 FIGURE1(Y)=X*6:'Line length = 6*X/256 units
50 FIGURE1(Y+1)=X+256:'Rot = X, entry code = 1
55 Y=Y+2
60 NEXT:'      Continue until figure completed
70 FIGURE1(Y+1)=0:'0 entry to terminate list
75 ' Draw it!
    FOR I = 0 TO 255 STEP 16:ROT=I:'Rotate figure
80 DRAW SET @64,23 USING FIGURE1(0)
    DRAW RESET @64,23 USING FIGURE1(0)
    NEXT I:A$=WINKEY$
```

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". To demonstrate the rotation available with "ROT", the figure is reset immediately after being drawn.

# END

# END

This statement is used to terminate your program.

Compiler BASIC and Interpreter BASIC	
END	Statement

Under Interpreter BASIC, "END" simply causes the program to cease running. Such a program may be continued by using the immediate "CONT" command.

Under Compiler BASIC, END causes a transfer back to DOS via the @EXIT address.

Compiler BASIC provides the ability to terminate a running program and automatically execute a DOS command. Note that this is not the same as RUN"program"; it is similar to DOS 5 BASIC's CMD"I","command". The syntax to use for this construct is:

**END"command string"**

For example, you could terminate the compiled program and invoke a Job file with a BASIC statement such as:

**END"DO MYFILE (A=LIST)".**

Any open files will be automatically closed prior to terminating the compiled BASIC program.

**EOF****EOF**

EOF determines if "End of File" has been encountered. The syntax for "EOF" is:

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>EOF(bufnum)</b>	<b>Function</b>
<b>bufnum</b>	Is the buffer number used to open the file.

This function is used to determine if the end of file has been reached when inputting from an open disk file. It is used primarily in conjunction with sequential files, but can also be used with random files. EOF is a function, and will return a "0" (FALSE) if the end of file has not been reached, or a "-1" (TRUE) if the end of file has been reached. It can be used with the "IF" statement, and will determine the outcome of the "IF", as it will return either a logical true or a logical false.

Assume that you have created a sequential file named MYDATA, and wish to access the information in it, but you do not know the amount of data in the file. The following program lines will illustrate how to use EOF to determine when the last piece of data has been accessed.

```
1000 OPEN "I", 1, "MYDATA"
1100 IF EOF(1) THEN PRINT "All data has been
accessed":END
      xxxx
      xxxx 'lines used to input and process data
      xxxx
1500 GOTO 1100
```

Notice that the "EOF" command is used prior to inputting any information. This will ensure that you will not try to input from an empty file, or after the end of file has been encountered. Either case would result in an "Input past end" error.

# ERASE

# ERASE

This DOS 6 interpreter BASIC statement is used to eliminate one or more arrays from a program. It's syntax is:

DOS 6 Interpreter BASIC	
<b>ERASE array[,...]</b>	Statement
<b>array</b>	The name of an array to delete.

Under DOS 6 Interpreter BASIC, arrays which are no longer needed may be eliminated to conserve memory space. Use the **ERASE** command for this purpose. For example,

```
10 OPTION BASE 1
20 DIM BIGARRAY(2000)
.
.
.
500 ERASE BIGARRAY
510 DIM BIGARRAY(1000)
```

eliminates the array, BIGARRAY, then re-dimensions it to 1000 elements.

# ERL

# ERL

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

Compiler BASIC and Interpreter BASIC		
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 number in which the error happened.

## ERR

## ERR

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

Compiler BASIC and Interpreter BASIC		
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.

The *Technical Information* section of this *Reference Manual* contains the error codes returned by "ERR".



# ERROR

# ERROR

This BASIC statement is used for runtime program error control.

Compiler BASIC and Interpreter BASIC	
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 /nnnn", or some similar message, 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.

## **ERRS\$**

## **ERRS\$**

This DOS 6 Interpreter BASIC function obtains, as a string, the DOS error message of the last *DOS error* generated. It's syntax is:

<b>DOS 6 Interpreter BASIC</b>		
<b>ERRS\$</b>	There is no operand!	Function

**ERRS\$** obtains the error message string which DOS would normally present for a DOS-related error; the string includes the DOS error number. For example, consider the following program:

```
10 ON ERROR GOTO 1000
20 OPEN "I",1,"TEST/DAT:7"
30 ...
1000 PRINT ERRS$
1010 STOP
```

If the TEST/DAT file did not exist, the error trapping routine starting at line 1000 would print,

```
24-File not in direclory
Break in 1010
```

# EXISTS

# EXISTS

EXISTS is a Compiler BASIC function which will check for the availability of the designated filespec.

## Compiler BASIC

**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.

```
10 PRINT"Enter the file to look for > ";
20 INPUT F$
30 IF EXISTS(F$) THEN PRINT"Found file"
   ELSE PRINT"File not found"
40 GOTO 10
```

"EXISTS" can be simulated with Interpreter BASIC as illustrated in the following example.

```
1000 ON ERROR GOTO 1050
1010 OPEN "I",1,F$
1020 PRINT F$;" EXISTS":CLOSE 1
1030 ON ERROR GOTO 0:RETURN
1050 PRINT F$;" DOES NOT EXIST"
1060 RESUME 1030
```

# EXP

# EXP

This function obtains the exponential of its argument.

Compiler BASIC and Interpreter BASIC	
<b>EXP(exp)</b>	Function
<b>exp</b>	Is a numeric expression.

"EXP(exp)" is equivalent to 2.7182818... raised to the *exp*'th power. This is the inverse of the natural log function; i.e.  $N = \text{EXP}(\text{LOG}(N))$ . If you're not familiar with this random-looking number, it pops up all over the place in engineering, scientific, and business problems.

Under Interpreter BASIC, "EXP" returns a single precision result. Compiler BASIC returns, with full precision, a value of the same type given.

"EXP(A)" will return an Overflow/Underflow error if "A" is out of bounds.

## FIELD

## FIELD

The **FIELD** statement is used to partition the buffer associated with a random access file, a type "R" file, and assign the segments of its record buffer to strings.

### Compiler BASIC and Interpreter BASIC

**FIELD** *bufnum*,*exp* **AS** *var\$*[,*exp2* **AS** *var2\$*] **Statement**

<b>bufnum</b>	Is the buffer number used in the associated <b>OPEN</b> statement. It may be a constant, or a numeric expression. <i>Bufnum</i> must be in the range of 1 to the total number of files allocated when entering BASIC or via <b>ALLOCATE</b> , inclusive, and must correspond to an open file.
<b>exp</b>	Are numeric constants or expressions denoting the maximum length (in bytes) of the fielded variable.
<b>var\$</b>	Are intermediate variables used to retrieve information from and pass information to the buffer. They must be string variables.

The **FIELD** statement is used to partition the buffer associated with an open random file. This partitioning allows you to divide a record up into fields, where each field denotes a particular piece of information in that record. The fielding of a record determines the length of each piece of information in the record, and where this information will physically reside in the record.

The value of these numeric constants or expressions must be in the range of 0 to 255, inclusive (0-256 for Compiler BASIC). If denoted as numeric expressions, these values must be enclosed within parentheses.

When information passes between the computer and the disk, a buffer is used as a temporary storage place for this information. Information is placed in this buffer with the "LSET" and "RSET" commands. Where this

information is physically placed in the buffer is determined by the "FIELD" statement.

The field statement will allow you to divide up the buffer into various "slots", assigning a variable name to each of these slots. When information is placed into or accessed from the buffer, it is done so by using the variable name which was assigned to each slot in the "FIELD" statement. The length of each of these slots is also determined by the "FIELD" statement. The total number of bytes to be fielded in a record must be less than or equal to the number of bytes that a record will contain.

The following example will illustrate how the "FIELD" statement is used.

Suppose that you wish to deal with a file that will contain records whose lengths will be 100 bytes. In each record, there will be 4 pieces of information (fields). Field 1 will be 20 characters long, and will represent the name of a person. Field 2 will be 10 characters long, and will represent an account number. Field 3 will be 30 characters long, and will represent address information. Field 4 will be 40 characters long, and will represent an account description. The following "OPEN" and "FIELD" statements will allow you to open such a file and field the buffer accordingly.

```
OPEN"R",1,"MYFILE/DAT",100
FIELD 1,20 AS NA$,10 AS AC$,30 AS AD$,40 AS DE$
```

Using the above lines in a program will produce the following results. A file by the name of MYFILE/DAT will be opened, and records in this file will have a length of 100 bytes. A buffer for this file will be set up in memory. The first twenty bytes of this buffer will represent name, and will be referenced by the variable NA\$. The next 10 bytes of this buffer will represent the account number, and will be referenced by the variable AC\$. The next thirty bytes will represent the address, and will be referenced by the variable AD\$. The last forty bytes will represent the description, and will be referenced by the variable DE\$.

More than one field statement may correspond to the same buffer. Variable names used in a "FIELD" statement may only be used to pass information to or retrieve information from the buffer. Using fielded variables for any other purpose will break the link between the variable and the buffer, and the variable will not be connected to the buffer until the original "FIELD" statement is re-executed. For more information on passing

information to and retrieving information from the disk, see OPEN, GET, PUT, LSET, RSET, MKI\$, MKS\$, MKD\$, CVI, CVS and CVD.

Under Compiler BASIC, "FIELD" is used with "R" type files; "X" type files use the corresponding "XFIELD" statement. "FIELD" partitions 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. String array elements are permissible in the "FIELD" statement.

Compiler BASIC supports a field length of 256. The corresponding "OPEN" statement should not include a *reclen* argument; the default is a record length of 256. This allows you to field a 256-byte record in a single string variable as illustrated in the next example.

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.

The compiler BASIC "FIELD" statement will generate a Field overflow error if the total length of the combined fields exceeds the *reclen* established by the "OPEN" statement.

### 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 compiler BASIC 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.

# FIX

# FIX

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

Compiler BASIC and Interpreter BASIC	
<b>FIX(exp)</b>	Function
<b>exp</b>	Is a numeric expression.

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



## FOR ... NEXT

## FOR ... NEXT

These statements implement the typical “FOR ... NEXT” loop construct.

Compiler BASIC and Interpreter BASIC	
<b>FOR</b> indexvar = start TO end [STEP step]	Statement
<b>NEXT</b> [indexvar1[,indexvar2...]]	Statement
<b>indexvar</b>	Is a loop index variable.
<b>start</b>	Is any numeric expression; the initial value of the loop index variable.
<b>end</b>	Is any numeric expression; the terminating top or bottom limit value of the loop.
<b>step</b>	Is any numeric expression; added to the loop variable in each iteration. Step may be negative, in which case <i>end</i> is a 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, in most cases 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.)

Under DOS 6 Interpreter BASIC, if the *start* value of the index exceeds the *end* value (or vice versa for negative STEPs), then the body of the "FOR ... NEXT" will be skipped! Also, DOS 6 Interpreter BASIC permits one and only one "NEXT" for each "FOR"!

The desired loop variable(s) can be specified after a "NEXT" statement. This is not necessary, however, as a "NEXT" without a designated index variable will *close* the most recent "FOR".. For instance, line 40 in the example program could simply be: **NEXT:NEXT**.

Compiler BASIC allows double precision variables 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.

## **FRE**

## **FRE**

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

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>FRE(exp)</b>	<b>Function</b>
<b>exp</b>	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".

Under DOS 6 Interpreter BASIC, the value returned is the amount of free memory.

# FUNCTION

# FUNCTION

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

Compiler BASIC	
<b>FUNCTION</b> name(input var list)	Statement
statements	
<b>ENDFUNC</b>	Statement
<b>input var</b>	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. **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 Compiler BASIC 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

# GET

GET reads a specified disk file record into a record buffer.

## Compiler BASIC and Interpreter BASIC

GET bufnum,recnum	Statement C
GET bufnum[,recnum]	Statement I

**bufnum** Is file control block buffer number, 1-15; it may be a numeric constant or numeric expression.

**recnum** Is the record number to read or write; it may be a numeric constant or numeric expression.

GET and PUT are the two type "R" disk file manipulation commands, and type "X" for Compiler BASIC. PUT writes the contents of the record buffer to the specified record in the specified currently open file. GET is used to retrieve information from a random file. The information that is retrieved is stored in the buffer that was used to open the file. Note that the "recnum" operand is mandatory in compiler BASIC; optional in Interpreter BASIC.

Under Interpreter BASIC, if the record number, *recnum*, is not specified, the computer will increment the current record number by one, after which it will perform a "GET" of the current record number. If no current record number has been established, the computer will perform a "GET" of record number one, and the current record number will be set equal to one.

Suppose you have opened a file and fielded the corresponding buffer. The buffer number used is 3. After executing one of the following statements:

```
GET 3,17
N%=2:N1%=16:GET N%+1,N1%+1
```

record 17 of the file will be contained in the designated buffer, and information dealing with this record may now be accessed by referencing the variables used in the "FIELD" statement.

## GOSUB

## GOSUB

This command allows your program to invoke unconditional program subroutine calls.

Compiler BASIC and Interpreter BASIC	
<b>GOSUB addr</b>	Statement
<b>RETURN</b>	Statement
<b>addr</b> Is a line number (or a Compiler BASIC 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 Compiler BASIC "POP" command documented elsewhere. The following table describes the possible errors which 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 in compiler BASIC
20  FOR X=0 TO 10: A(X)=RND(X) : B(X)=RND(0) :
    ?A(X),B(X):NEXT
30  GOSUB"SORTA"
40  GOSUB"PRINTA":' Could be GOSUB 140
50  GOSUB"SORTB"
60  GOSUB"PRINTB"
70  END
80  '
100 "SORTA":' Alternatively: JNAME"SORTA"
```



```
110 SCLEAR:KEY A(0):TAG B(0):SORT 11:RETURN
120 "SORTB"
130 SCLEAR:KEY B(0):TAG A(0):SORT 11:RETURN
140 "PRINTA"
150 FOR X=0 TO 11: PRINT A(X),B(X):NEXT:RETURN
160 "PRINTB"
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.

Note that in the example, lines 150 and 170 illustrate the runtime error handling of "subscript out of bounds". To correctly run the program, change the '11' to '10' in both of those lines.

# GOTO

# GOTO

This statement allows your program to invoke unconditional program branching.

Compiler BASIC and Interpreter BASIC	
<b>GOTO addr</b>	Statement
<b>addr</b>	Is a line number (or a Compiler BASIC label).

**GOTO** is the standard BASIC way to transfer program execution to just about any desired point in the program. A conventional line number may be used in Interpreter BASIC; a line number or a label can be specified in Compiler BASIC

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.

## **&H**

## **&H**

This function indicates that the argument is a hexadecimal number rather than a decimal format number.

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>&amp;Hd0...d4 Hexadecimal number</b>	<b>Function</b>

To represent a number in its hexadecimal format, you may use the characters “&H” as a prefix to the number. This may be useful when you wish to define an address for a user machine language subroutine (see “DEFUSR”).

One to four hexadecimal digits may follow the “&H” prefix. Hexadecimal digits consist of the numeric digits “0-9”, as well as the alphabetic letters “A-F” which indicate the numeric values “10-15”. The number represented using the “&H” prefix will always be taken as two’s complement notation. For example, the assignments:

**A = B AND &H100**

and

**A = B AND 256**

are equivalent. Other examples are:

**A=&H11**

A would be set equal to the decimal number 17.

**A=&HA9**

A would be set equal to the decimal number 169.

**A=&HF000**

A would be set equal to the decimal number -4096.

# HEX\$

# HEX\$

This function converts numeric expressions to strings of hexadecimal digits.

<b>Compiler BASIC and DOS 6 Interpreter BASIC</b>	
<b>HEX\$(exp16)</b>	<b>Function</b>
<b>exp16</b>	Is in the range <-32768 to 32767>

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

## IF THEN ELSE

## IF THEN ELSE

These statements implement the typical “IF...THEN...ELSE” conditional structure.

Compiler BASIC and Interpreter BASIC	
IF <i>cond</i> THEN action [ELSE default action]	Statement
IF <i>cond</i>	Statement C
program code	
[ELSE	Statement C
program code]	
ENDIF	Statement C

“IF ... THEN ... ELSE” comprise the critical conditional execution statements. The *cond* is evaluated; if it is TRUE, then the action statements are executed; if the *cond* is FALSE, then the default action statements following the “ELSE” are executed, if any. If no “ELSE” statement is provided, execution continues with the next program line.

Compiler BASIC 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

# INC

INC is a Compiler BASIC statement used to increment an integer variable.

Compiler BASIC	
INC 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

## INKEY\$

## INKEY\$

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

Compiler BASIC and Interpreter BASIC		
INKEY\$	There is no operand!	Function

INKEY\$ returns a null string (i.e. zero length) 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$=INKEY$:IF A$="" THEN 20
30 IF A$ = "X" THEN PRINT"Exiting program":END
40 PRINT ASC(A$);
50 GOTO 20
```

This program will display the ASCII value of the key pressed until you either <BREAK> the program or press a capital "X".



# INP

# INP

This function obtains the value of the specified CPU port.

Compiler BASIC and Interpreter BASIC	
<b>INP(portnum)</b>	Function
<b>portnum</b>	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 a specified CPU I/O port.

**Programmer's Note:**

Under Interpreter BASIC, the value of *portnum* can actually range from 0 through 32767; the actual value used for the port determination will be “portnum modulo 256”. Thus, a portnum of 256 designates port 0; a *portnum* of 257 designates port 1. However, the high-order value will be presented to the high-order address lines of the CPU address bus. This may be useful in addressing internal and external ports of the 64180/Z180 processor which requires a high-order zero value to reference internal ports and a high-order non-zero value to reference external ports.

# INPUT

# INPUT

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

## Compiler BASIC and Interpreter BASIC

INPUT [ <i>@pos</i> ,][ <i>"string"</i> ]; <i>var1</i> [, <i>var2</i> ...]	Statement	C, 5
INPUT [ <i>"string"</i> ];, <i>var1</i> [, <i>var2</i> ...]	Statement	6

<b>@pos</b>	When specified, positions the video cursor to <i>pos</i> .
<b>string</b>	When specified, the character string is displayed prior to the input.
<b>var</b>	Is any appropriate variable.

INPUT displays a question mark, then reads data from the keyboard. An optional "prompt" string may be printed prior to the "?". Leading blanks are skipped while reading. Strings (string variable specified) are read until a comma or an <ENTER> "CHR\$(13)" is reached. Numbers (numeric variable specified) are read until a space, a comma, or an <ENTER> is encountered.

DOS 5 Interpreter BASIC and Compiler BASIC allow you to establish the video screen cursor location prior to the input or message string display. The "@pos" is identical in concept to the "PRINT @pos". See additional information under "INPUT@".

Under DOS 6 Interpreter BASIC, the optional prompt string may be terminated with either a semi-colon or a comma. If a comma is used, then the automatic question mark display is suppressed.

```
10 INPUT "Enter your weight in pounds";p
20 PRINT "Your weight in kilograms would be ";
   p*0.45359237
```

## **INPUT#**

## **INPUT#**

The **INPUT#** statement is used to retrieve information from a sequential file into variable(s). The syntax used with the “**INPUT#**” command is:

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>INPUT#bufnum, var1 [,var2 ...]</b>	<b>Statement</b>
<b>bufnum</b>	Is a file control block buffer number, <1-15>, used to open the file.
<b>var</b>	Is any appropriate variable used to store the information retrieved.

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

Sequential files are created by specifying an **OPEN“O”** or **OPEN“E”** command, followed by one or more **PRINT#** commands. After a sequential file has been created, the information in it may be accessed by using the **OPEN“I”** and **INPUT#** commands. The “**INPUT#**” command can be thought of as performing a function similar to the “**INPUT**” command, the exception being that the information is not entered from the keyboard. Rather, it is retrieved from the disk. Like the “**INPUT**” command, “**INPUT#**” can only be executed from within a program, and cannot be executed from the BASIC Ready prompt.

The variable types used in an “**INPUT#**” statement must be the same type of variable used when the information was written to the file via the “**PRINT#**” command. At least one variable must be specified with the “**INPUT#**” command. If multiple variables are specified with the “**INPUT#**” command, they must be separated by commas.

After execution of an “**INPUT#**” command, the variable(s) specified will be assigned values corresponding to the data retrieved from the disk. If

you try to execute an "INPUT#" command after all of the data has been retrieved from the file, an input past end error will be generated.

Suppose a file called MYFILE/SEQ was created using the OPEN"O" and PRINT# commands, and this file contains the following pieces of data:

JONES  
THOMAS  
12  
MALE

The following commands may be used to access this information:

```
OPEN" I", 1, "MYFILE/SEQ"  
INPUT#1, LN$, FN$, AG%  
INPUT#1, SE$
```

After the execution of the first two commands, the file MYFILE/SEQ would have been opened for sequential input, the variable "LN\$" would have been assigned the value "JONES", the variable "FN\$" would have been assigned the value "THOMAS", and the variable "AG%" would have been assigned the value "12". Note that the last piece of data in the file ("MALE") would not have been accessed by either of the first two commands. However, after the third command "INPUT#1,SE\$" has been executed, the variable "SE\$" would be assigned a value of "MALE".

"INPUT#" deals with data in a disk file in a special way. For more information on creating sequential files that are accessed by the "INPUT#" command, refer to "OPEN" ("O", "E" and "I") and "PRINT#".

# INPUT\$

# INPUT\$

This DOS 6 interpreter BASIC statement is used to input a string of characters from either the keyboard or a disk file. It's syntax is:

DOS 6 Interpreter BASIC		
INPUT\$(intvar[,bufnum])		Statement
<b>intvar</b>	Is the quantity of characters to accept; intvar must be in the range <1-255>.	
<b>bufnum</b>	Is an optional file buffer number; if bufnum is omitted, the input is fetched from the keyboard.	

**INPUT\$** accepts a designated quantity of characters from either the keyboard or a previously opened disk file. The characters are read into a string variable. The input stops as soon as the designated quantity of characters is fetched. During the input, the characters fetched are not displayed on the video screen.

## INPUT@

## INPUT@

This DOS 5 BASIC statement provides a facility for controlling keyboard input to a string variable with prompting and screen field highlighting. The expanded form of **INPUT@** is:

DOS 5 Interpreter BASIC	
<b>INPUT@pos[, "message"], fw, "\$ #[*](f)"; var\$</b>	Statement
<b>pos</b>	The screen position for the message or field.
<b>var</b>	Variables used to store the data retrieved.
<b>fw</b>	The maximum input field width <1-240>.
<b>\$</b>	Acceptance of characters <32d-127d>.
<b>#</b>	Acceptance of characters <0-9, period, minus, plus>.
<b>*</b>	Specifies immediate <ENTER> on maximum input.
<b>f</b>	Designates a field fill character ["_"].
<b>var\$</b>	The string variable to receive the input.

The expanded form of **INPUT@** allows you to specify an input field width, an input field fill character, whether the input should be alphanumeric or just numeric, and whether the input should automatically terminate when the "field width" number of characters have been entered rather than requiring a hard <ENTER>.

The single input variable will always be a string variable and the input, regardless of designated type, will always be a string of characters. Note that since the "immediate <ENTER> on maximum input" character is examined before the programmed fill character, you cannot designate an asterisk as the fill character unless you specify the forced immediate <ENTER>. Thus, **"#\*"** will be interpreted as accepting numeric only, immediate <ENTER> on full field, and use an asterisk as the field fill character. On the other hand, **"#"** will default the field fill character to an underline.

# INSTR

# INSTR

This function will search a string for a designated substring.

Compiler BASIC and Interpreter BASIC		
INSTR([exp,] exp1\$,exp2\$)		Function
exp1\$	Is the string to search.	
exp2\$	Is the string to search for within the target string.	
exp	Is an optional search start point.	

The **INSTR** command allows you to search for a specified sub-string within a given target string. “INSTR” returns the position number in the target string of where the sub-string was found. If the substring was not found, “INSTR” returns a numeric value of “0”.

The starting position may be either a numeric constant or a numeric expression, and must represent an integer value in the range of 1 to 255, inclusive. The target string and sub-string may be either string constants or string expressions.

“INSTR” will begin the search of the target string for the sub-string from the starting position specified (if no starting position is specified, “INSTR” will begin the search from the first character of the target string), and will return a numeric value corresponding to the position in the target string of where the first occurrence of the sub-string is found. If the sub-string is not found in the target string, “INSTR” will return a 0. If the sub-string to be searched for is a null string, “INSTR” will return the starting position of the search, as the null string is a sub-set of any string.

Other occurrences may cause “INSTR” to return a zero. They are:

- If the target string is a null string.

- If the starting position is a number greater than the length of the target string.

Suppose you have the following lines in a program:

```
A$="ROY IS A BOY":  
B$="OY":C$="ROY":D$="oy":E$="ROYIS"  
A%=INSTR(A$,C$)  
B%=INSTR(2,A$,B$)  
C%=INSTR(3,A$,B$)  
D%=INSTR(2,A$,C$)  
E%=INSTR(A$,D$)  
F%=INSTR(A$,E$)
```

After executing the above lines, the following variables will have been assigned these values:

**A%=1 B%=2 C%=11 D%=0 E%=0 F%=0**

Note that the value of E% will be 0. This is because the sub-string ("oy") is in lower case, and there are no lower case letters in the target string. Also note that the value of F% will be 0. This is because the string "ROYIS" does not appear in the target string (there is a space between the words "ROY" and "IS" in the target string).

#### Example Programs:

```
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
```

```
CLS  
20 A$=WINKEY$  
PRINT "a$=";ASC(A$)  
IF INSTR(CHR$(8)+CHR$(9)+CHR$(13)+CHR$(10),A$)  
THEN PRINT ASC(A$)  
FOR I=1 TO 1000:NEXT I:GOTO 20
```



# INT

# INT

This is the “greatest integer” function.

Compiler BASIC and Interpreter BASIC	
<b>INT(exp)</b>	Function
<b>exp</b>	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**

# **INVERT**

# **INVERT**

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

Compiler BASIC	
INVERT	Statement

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

# JNAME

# JNAME

This Compiler BASIC statement is used to establish a line label.

Compiler BASIC		
JNAME"label"		Statement
"label" [EQU value]		Statement
label	Is a (unique) string of characters representing a memory location.	
value	Is an optional value assigned to the label.	

Labels are used to establish branch points for use with "GOTOs", "GOSUBs", or any BASIC statement.

Line labels must **not** have imbedded spaces. The labels must be entered as a contiguous string of non-space characters. For instance, "PRINTA", "SORTA", "PRINTB", and "SORTB" are all examples of valid labels.

All BASIC (HIGH-MODE) referenced labels must be located within the HIGH-MODE portion of your program. A label defined in the Z80-MODE of your program can be referenced by the assembler portion only. See the use of "ENLOOP" and "ENDENC" on page 128.

The "EQU" operator of "label" allows you to define values for labels to be typically used in conditional compilation. For example, the following short program illustrates using a label value along with Compiler BASIC's "\*IF" compiler directive to select alternative program statements for compilation:

```
"DOS5" EQU 0:"DOS6" EQU 1
*IF DOS5
PRINT"DOS 5"
*ENDIF
*IF DOS6
```

```
PRINT "DOS 6"  
*ENDIF
```

If two or more labels are defined with the **JNAME**"label" or "label" statement, a "Multiply defined symbol" error will be issued.

**Example Program:**

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

Note that in the example, lines 150 and 170 illustrate the runtime error handling of "subscript out of bounds". To correctly run the program, change the "11" to "10" in both of those lines.

# LEFT

# LEFT

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

Compiler BASIC	
LEFT	Statement C

This compiler BASIC statement scrolls the entire screen left by one character. The entire last screen column is cleared, and all of the 0th column is written over with the 1st column.

# LEFT\$

# LEFT\$

This function extracts a sub-string of a string.

Compiler BASIC and Interpreter BASIC	
<b>LEFT\$(exp\$,exp1)</b>	Function
<b>exp\$</b>	Is any string expression.
<b>exp1</b>	Is the number of leftmost characters to use for the obtained substring.

**LEFT\$** extracts a sub-string 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  $\text{len}(\text{exp\$}) \geq \text{exp}$ .

# LEN

# LEN

This function obtains the length of its string argument.

Compiler BASIC and Interpreter BASIC	
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

# LET

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

Compiler BASIC and Interpreter BASIC	
<b>[LET] var = exp</b>	Statement
<b>var</b>	Is any variable.
<b>exp</b>	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 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. Compiler BASIC 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

# LINEINPUT

LINEINPUT is used to accept keyboard input into a string.

## Compiler BASIC and Interpreter BASIC

LINEINPUT [@pos] ["string";] var\$	Statement C
LINE INPUT ["string";] var\$	Statement I

**@pos**      An optional screen position to initiate the video cursor prior to input.

**string**      Is an optional message to be displayed prior to accepting the input.

**var\$**      Is any appropriate string variable.

Note: the space between "LINE" and "INPUT" is optional under DOS 5 Interpreter BASIC; mandatory under DOS 6.

The **LINE INPUT** command is very similar to the "INPUT" command. It will allow you to enter information from the keyboard to be stored in a variable. The differences between the "LINEINPUT" command and the "INPUT" command are as follows:

- No question mark will appear when the input is taken.
- Only one variable may be assigned a value.

All characters entered before <ENTER> is pressed will be assigned to the variable specified (i.e. commas and quotes may be input from the keyboard, and leading spaces are **not** ignored).

An optional "prompt" string may be printed. The "LINE INPUT" statement turns **on** the cursor prior to performing the input and turns **off** the cursor upon concluding the input.

The prompting message is optional; if used, it must be included within quotes, and must be separated from the variable by a semicolon. If the

prompting message is not used, a semicolon cannot be used. As is the case with the "INPUT" command, "LINE INPUT" cannot be issued from the BASIC Ready prompt.

Suppose that you wish to input a person's name and title into a program, and you wish to separate the name from the title by use of a comma. Using the "LINE INPUT" command, you may now input the comma from the keyboard to be taken as part of the input. The following "LINE INPUT" command may be used to accomplish this.

```
100 LINE INPUT"Enter Name, Title";A$
```

When the computer executes the above command, you will see the prompt "Enter Name, Title" appear, and there will be no question mark after the prompt. The computer will now be awaiting your input. If you answer this prompt by typing in the response "JOHN JONES, PRESIDENT", "A\$" will be assigned all characters that you have typed in, prior to pressing the <ENTER> key.

## LINEINPUT#

## LINEINPUT#

LINE INPUT# is used to read from a sequential file into a string variable.

### Compiler BASIC and Interpreter BASIC

LINEINPUT#bufnum, var\$	Statement C
LINE INPUT#bufnum, var\$	Statement I

**bufnum** Is a file control block buffer number, <1-15>, used when the file was opened.

**var\$** Is any string variable used to stored the retrieved information.

Note: the space between "LINE" and "INPUT" is optional under DOS 5 Interpreter BASIC; mandatory under DOS 6.

LINE INPUT# reads a string from an "I" type file into a variable. All characters starting at the current read point up to a <ENTER> "CHR\$(13)" or the end of file are read into the string, up to the limit of 255 characters. It functions similarly to the "LINE INPUT" command, with the exception being that the input is taken from the disk, rather than the keyboard.

"LINE INPUT#" differs from "INPUT#" in several ways. As noted in the PRINT# command, INPUT# will read information in from the disk until it encounters a comma, a carriage return, the end of file, or the 255th character when dealing with string information. When using LINE INPUT#, commas will not be taken as delimiters of the string, and hence may be included in the input from disk. The "LINE INPUT#" of a variable will terminate when a carriage return, the end of file, or the 255th character of a string is encountered. As is the case with "INPUT#", "LINE INPUT#" cannot be executed from the BASIC Ready prompt.

Assume the following data is stored in a disk file, and the file has been opened using buffer number 1.

**JOHN JONES , PRESIDENT , ABC CORPORATION<ENTER>**

If the command **LINE INPUT#1,A\$** is used to input the above information, "A\$" would be assigned the value:

**JOHN JONES , PRESIDENT , ABC CORPORATION**

Realize that all of the characters (including the commas) would be read in and assigned to A\$.

If the command **INPUT#1,A\$** were used instead of "LINEINPUT#", the value of A\$ would be "JOHN JONES", as "INPUT#" will read information until it encounters a comma. For more information on how data is stored on the disk in a sequential file, see "PRINT#".

# **LINESPAGE**

# **LINESPAGE**

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

## **Compiler BASIC**

**LINESPAGE = 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. Its use is similar to using the DOS FORMS filter with "Lines=exp,Ffhard"; however, Compiler BASIC's paging control is strictly internal to BASIC.

For additional printer paging statements, see "LMARGIN", "PAGELEN", "PZONE", and "RMARGIN".

# **LMARGIN**

# **LMARGIN**

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

<b>Compiler BASIC</b>	
<b>LMARGIN = exp</b>	<b>Statement</b>
<b>exp</b>	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. Its use is similar to using the DOS FORMS filter with "Margin=exp"; however, Compiler BASIC's paging control is strictly internal to BASIC.

For additional printer paging statements, see **LINESPAGE**, **PAGELEN**, **PZONE**, and **RMARGIN**.

# LOAD

# LOAD

This statement will load a "CMD-type" program from disk into memory.

Compiler BASIC	
LOAD"filespec\$"	Statement
filespec\$ Designates the file to load.	

LOAD loads a machine language program from disk into memory without executing it. It is identical to DOS 5 Interpreter BASIC's CMD"L" command.

The Interpreter BASIC "LOAD" statement, which loads a BASIC program into memory, is discussed in the section on *Editing Interpreter BASIC*.

# LOC

# LOC

The LOC command is used primarily with random files, and will return a value corresponding to the current record number of the given file. The syntax for the "LOC" command is:

## Compiler BASIC and Interpreter BASIC

**LOC(bufnum)**

**Function**

**bufnum** Represents the buffer number used to open the file in question. *Bufnum* may be either a numeric constant or a numeric expression, and must correspond to an open file.

When a file is in an open state, the computer maintains some control information dealing with that file. One piece of information that is available to the user is the record number currently being dealt with. The LOC command will return the current record number that the computer has accessed. If no record in an open file has been accessed, "LOC" will return the value 0.

Suppose you have opened a file using buffer number 2, and have fielded the buffer accordingly. If the following commands are executed:

```
GET 2,17
```

```
A%=LOC(2)
```

the variable "A%" will be assigned the value 17.



## LOF

## LOF

The LOF command is used primarily with random files, type "R", and will return a value corresponding to the last record number of the given file. The syntax for the "LOF" command is:

**Compiler BASIC and Interpreter BASIC****LOF(bufnum)****Function**

**bufnum** Represents the buffer number used to open the file in question. *Bufnum* may be either a numeric constant or a numeric expression.

The LOF command provides a means of determining the number of records that have been written to a random file. Note that if a file has been pre-created using the DOS "CREATE" command, "LOF" will return a number corresponding to the highest record number actually written to, not the number of records that have been pre-created.

Suppose you have a file named MYFILE/DAT, and the highest record number written to is record number 43. If the file has been opened using buffer number 3, and has been fielded accordingly, the following command will result in the variable A% being set equal to 43.

```
A%=LOF(3)
```

Under Compiler BASIC, LOF() restricts its use on all file types except "R". The "X" type file does not maintain an end-of-file pointer which is valid for the last record.

```
ALLOCATE 1:OPEN "R",1,"testfile/dat:4",20
FIELD 1, 16 AS A$, 2 AS A1$, 2 AS A1$
LSET A$="This is record: "
FOR I = 1 TO 45:A1$ = MKI$(I):PUT 1,I:NEXT
CLOSE:SYSTEM"dir test:4"
OPEN "R",1,"testfile/dat:4",20
PRINT LOF(1):CLOSE:END
```

# LOG

# LOG

This function obtains the natural logarithm of its argument.

Compiler BASIC and Interpreter BASIC	
<b>LOG(exp)</b>	Function
<b>exp</b>	Is a numeric expression.

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

**LOG(A)** will return an illegal function call error if A is zero or negative.

# LPOS

# LPOS

This Interpreter BASIC function obtains the relative position of the line printer's print head relative to the line printer's program buffer.

## Interpreter BASIC

**LPOS (dummy)**

**Function**

**dummy** Is a dummy argument required for functions.

This function allows you to obtain the logical position of the printer's print head; i.e. where the next character LPRINT'd would be placed. For example, after the following program is RUN,

```
10 LPRINT "This is a message";  
20 PRINT LPOS (0)  
30 LPRINT TAB (5) ;  
40 PRINT LPOS (0)
```

the video screen would display,

```
run  
22  
5
```

# LPRINT

# LPRINT

This statement is used to print data to the printer.

## Compiler BASIC and Interpreter BASIC

LPRINT [ <i>item</i> ] [, ; TAB( <i>exp</i> )] ...]	Statement
---	-----------

LPRINT USING <i>format</i> ; <i>item</i>	Statement
--	-----------

<i>item</i>	Is a "stringliteral" or a numeric   string expression.
-------------	--

,;	Are delimiters.
----	-----------------

All LPRINT statements are equivalent to "PRINT" statements, but are directed to the line printer rather than the video screen. A comma delimiter or equivalently TAB(255) tabs the cursor to the next printer zone. A semi-colon delimiter retains the print position immediately following the output of *item*. By not specifying a terminator, the next "LPRINT" will occur on the next print line. You can tab to a particular column by specifying a "TAB(*exp*)" with *exp* equal to the column position.

Compiler BASIC "USING" is a string expression, and can thusly be manipulated just like any string. Compiled and interpreted BASIC "PRINT USING" statements usually produce the same output.

Compiler BASIC also allows you to alter the default screen or printer TAB positions with the "SZONE" and "PZONE" commands respectively documented elsewhere in this manual.

## LSET

## LSET

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

Compiler BASIC and Interpreter BASIC	
<b>LSET var\$ = exp\$</b>	Statement
<b>var\$</b>	Is a FIELDed string variable used in the FIELD statement that points to the location in the buffer where the information is to be added.
<b>exp\$</b>	Is the information that you wish to place in the buffer, and must be a string constant or a string expression.

The LSET command will allow you to place information in the buffer associated with a random file, prior to writing the information in the buffer out to disk. 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*).

When dealing with random files, the "FIELD" statement is used to set up and partition the buffer associated with the file. String variables are used in the "FIELD" statement to designate various slots for information storage and retrieval in the buffer. The "LSET" command allows you to place information in these slots in the buffer, prior to writing the information out to disk.

The "LSET" command will left-justify the information in the buffer. That is to say, if the length of the string to be placed in the buffer is less than the length allocated for the particular slot, trailing spaces will be inserted

at the end of the string in the buffer. This will make the string in the buffer the same length as specified in the "FIELD" statement.

If the length of the string to be "LSET" into the buffer is greater than the fielded length, the left most part of the string will be placed in the buffer, and any characters to the right of the total allocated space will be truncated. See "RSET" to right-justify a string into the buffer.

The commands "MKI\$", "MKS\$", and "MKD\$" are also used in conjunction with the "LSET" statement. Because the buffer is fielded in terms of string variables, only string values may be LSET into the buffer. The "MKI\$", "MKS\$", and "MKD\$" commands are used to change numeric data into compressed string representations of numbers, and will create strings of two bytes, four bytes, and eight bytes respectively. When performing an "LSET" using the "MKI\$", "MKS\$" or "MKD\$" commands, the length of the fielded variable to be "LSET" must be at least two bytes, four bytes, or eight bytes, respectively. For more information on commands that are used with "LSET", refer to the commands "MKI\$", "MKD\$", "MKS\$", and "FIELD", and the example below.

Suppose you have a file called MYFILE/DAT, and have opened the file to have record lengths of 45 bytes. In addition, assume that the buffer corresponding to the file (buffer number 1) has been fielded with the following statement, and the variables listed below have been assigned the given values:

```
FIELD 1, 31 AS NA$, 2 AS A2$, 4 AS A4$, 8 AS A8$  
NM$="JOHN JONES, PRESIDENT":  
A2%=92:A4!=23.79:A8#=123498.63
```

The LSET statements you may use to place these values into the buffer may look like this:

```
LSET NA$=NM$  
LSET A2$=MKI$(A2%)  
LSET A4$=MKS$(A4!)  
LSET A8$=MKD$(A8#)
```

The values of the variables A2%, A4!, and A8# will be stored in the slots in the buffer pointed to by the variables A2\$, A4\$, and A8\$, respectively. They will be stored as compressed string representations of the values the variables have been assigned.

The value of NM\$ will be stored in the slot in the buffer pointed to by the variable NA\$. Realize that since the length of NM\$ is 21 characters, the last 10 characters of the slot in the buffer pointed to by NA\$ will be spaces (CHR\$(32)). If the length of NM\$ would have been longer than 31 characters, the left-most 31 characters would have been placed in the buffer, and the remaining characters would have been truncated (in essence, ignored).

The "LSET" command will typically be used prior to performing a data write to a random file. For more information on performing a data write to a random file, see "OPEN", "FIELD" and "PUT".

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" or "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" or "RSET" commands, work on any string variable, not just FIELDed string variables.

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

```
LSET A$="HELLO":'  
LSET A$="12345678912":'  
RSET A$="HELLO":'  
LSET A$=MKD$(1.2345#):'
```

```
Now A$="HELLO  "  
Now A$="1234567891"  
A$="  HELLO"  
Now first 8 bytes of A$  
contain the floating point  
double precision number  
1.2345#
```

# MEM

# MEM

This function obtains the amount of free stack space or free memory.

<b>Compiler BASIC and Interpreter BASIC</b>		
---	--	--

<b>MEM</b>	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. Under DOS 6 Interpreter BASIC, the value returned is the amount of free memory.



## MERGE

## MERGE

The MERGE command will allow you to merge a program file stored on disk (in ASCII) with a program resident in memory, with the resultant program being stored in memory. The syntax for the "MERGE" command is:

Interpreter BASIC	
MERGE"filespec"	Statement
<b>filespec</b>	Is the program file, saved in ASCII, which is to be merged with the current program in memory. <i>Filespec</i> may be represented as a string constant or a string expression. If represented as a string constant, filespec must be contained within quotes.

The MERGE command will read in (line by line) the program from disk, and merge these lines in with the existing program. Any line number in the program to be merged that does not exist in the program in memory will be added to the program in memory. Any line number in the program to be merged that does exist in the program in memory will overwrite the line in memory.

The "MERGE" command provides for an easy way to merge subroutines which are common to several different programs into these programs without always having to type in the subroutine. The following example will illustrate how the "MERGE" command functions.

Suppose you have a program which is resident in memory, and this program consists of the following statements:

```
10 FOR L=1 TO 100
20 PRINT L
30 NEXT L
```

Assume also that you have a program named MYPROG/ASC stored in ASCII on disk, and this program consists of the following statements:

```
5 DEFINT A-Z
10 FOR L=1 TO 500
25 'THIS LINE HAS BEEN MERGED IN
40 GOTO 10
```

If you wish to merge the program MYPROG/ASC with the program currently in memory, you may do so by issuing the following command:

**MERGE"MYPROG/ASC"**

By giving the above command, the program resident in memory will be changed to the following:

```
5 DEFINT A-Z
10 FOR L=1 TO 500
20 PRINT L
25 'THIS LINE HAS BEEN MERGED IN
30 NEXT L
40 GOTO 10
```

Before merging in a program, you should make sure that there is enough free memory for the program to be merged in. Also, note that the "MERGE" command is usually issued from the BASIC Ready prompt. However, if incorporated within a program, the "MERGE" will be done, but execution of the program will cease.

# MID\$()=

# MID\$()=

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

## Compiler BASIC and Interpreter BASIC

**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 you wish to replace the specified portion of the current string with.

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 and allows you to perform a character for character replacement of any characters within a string. "MID\$" never changes the length of the string variable.

The "MID\$=" command will perform a character for character replacement on a given string with the replacement string. It may not be used to lengthen or shorten an existing string. If the optional length parameter, *exp2*, is not specified, the number of characters involved in the replacement will be determined by the length of the replacement string. If the length parameter differs from the length of the replacement string, one of several things may happen.

- If the length parameter is less than the length of the replacement string, the length parameter will take precedence, and only the left-most number of characters as specified in the length parameter will be changed.

- If the length parameter is greater than the length of the replacement string, the replacement string will take precedence, and only those characters specified in the replacement string will be changed.

If the parameters specified in the "MID\$=" command would cause the original string to become larger, only those characters up to the end of the original string would be changed, and the length of the string would remain unchanged. In essence, the extra characters at the end of the replacement string would be ignored.

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.

For still more examples, suppose you have a string variable A\$ set equal to the value "THIS IS IT". The following "MID\$=" commands would have these affects on A\$.

MID\$(A\$,3,2)="AT"	A\$ would change to "THAT IS IT"
MID\$(A\$,6,2)="WAS"	A\$ would change to "THIS WA IT"
MID\$(A\$,3,8)="AT'S IT"	A\$ would change to "THAT'S ITT"
MID\$(A\$,9,3)="ALL"	A\$ would change to "THIS IS AL"

# MID\$

# MID\$

This function extracts sub-strings of a string.

Compiler BASIC and Interpreter BASIC		
MID\$(exp\$,exp1 [,exp2])	Function	
exp\$	Is any string expression.	
exp1	Is the starting position.	
exp2	Is the optional substring length. If <i>exp2</i> is omitted, the rest of <i>exp\$</i> after <i>exp1</i> 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)**

and,

**RIGHT\$(exp\$,exp)**

is equivalent to

**MID\$(exp\$,len(exp\$)-exp+1)**

assuming len(exp\$) >= exp.

## MKD\$

## MKD\$

The MKD\$ command (MaKe Double precision string) will change a numeric value into an eight-byte string which is a compressed representation of the value. This command is used primarily with the "LSET" and "RSET" commands to place numeric data into the buffer associated with an open random file. The syntax for the "MKD\$" command is:

### Compiler BASIC and Interpreter BASIC

**MKD\$(exp)**

Function

**exp**

Is a numeric expression of the desired type

MKD\$ maps a double precision number to an eight-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 two-byte string for storing integers and "MKS\$" maps single precision numbers to four-byte string for storing single precision expressions. *exp* may be either a numeric constant or a numeric expression and can represent any value which may be assigned to a double precision variable. Up to 16 significant digits will be maintained. To convert an eight-byte compressed string representation of a number back to a numeric value, use the "CVD" command.

Since only strings may be stored in the buffer associated with an open random file, there exists a need to change numeric data into a string form. "MKD\$" provides a way to change numeric data into a string. The string formed by "MKD\$" will always be eight bytes in length, regardless of the actual value to be converted. The resultant string value obtained when performing an "MKD\$" command will be the compressed form of a number, contained in an eight-byte string. After a numeric value has been changed into an eight-byte compressed string, it may then be placed into a buffer via the "LSET" and "RSET" commands. (Note: This is not the same as the "STR\$" command, as "STR\$" produces an ASCII string, not a compressed string representation of a number.)

For example:

```
10  ALLOCATE 1: 'Needed for compiler BASIC
20  A#=1.2345678#: B=2.71828
30  OPEN "R",1,"TEST/DAT"
40  FIELD 1,8 AS PY$,4 AS E$
50  LSET PY$=MKD$(A#): LSET E$=MKS$(B)
60  PUT 1,1
70  CLOSE

.
.
```

The string-encoded contents of "A#" are "LSET" into the first eight bytes of the record buffer, effectively storing "A#", and "B" is stored in the next four 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.

Suppose you have opened and fielded a random file, and wish to place a double precision value into the buffer. The fielded variable you are dealing with is "A8\$", and the value you wish to place in the part of the buffer pointed to by "A8\$" is contained in the variable "A8#". The following command will cause an eight-byte compressed string representation of the value stored in "A8#" to be written to the portion of the buffer pointed to by "A8\$".

**LSET A8\$=MKD\$(A8#)**

Note that the fielded length of the variable "A8\$" must be at least eight bytes, and in most cases will be exactly eight bytes.

## MKI\$

## MKI\$

The MKI\$ function (MaKe Integer string) will change a numeric value into a two-byte string which is a compressed representation of the value. This command is used primarily with the "LSET" and "RSET" commands to place numeric data into the buffer associated with an open random file.

### Compiler BASIC and Interpreter BASIC

**MKI\$(exp)**

Function

**exp**

Is a numeric expression of the desired type

Since only strings may be stored in the buffer associated with an open random file, there exists a need to change numeric data into a string form. MKI\$ provides a way to change numeric data into a string. The string formed by "MKI\$" will always be two bytes in length, regardless of the actual value to be converted. The numeric value must be within the range of -32768 to +32767, inclusive. If the value is not an integer, any numbers to the right of the decimal point will be truncated. To convert a two-byte compressed string representation of a number back to a numeric value, use the "CVI" function.

The resultant string value obtained when performing an "MKI\$" command will be the compressed form of an integer, contained in a two-byte string. After a numeric value has been changed into a two-byte compressed string, it may then be placed into a buffer via the "LSET" and "RSET" commands. (Note: This is not the same as the "STR\$" command, as the "STR\$" command produces an ASCII string, not a compressed string representation of a number.)

Suppose you have opened and fielded a random file, and wish to place an integer value into the buffer. The fielded variable is "A2\$", and the value is contained in the variable "A2%". The command **LSET A2\$=MKI\$(A2%)** will cause a two-byte compressed string representation of the value stored in "A2%" to be written to the portion of the buffer pointed to by "A2\$". Note that the fielded length of the variable "A2\$" must be at least two bytes, and in most cases will be exactly two bytes.



## MKS\$

## MKS\$

The MKS\$ function (MaKe Single precision string) will change a numeric value into a four-byte string which is a compressed representation of the value. This command is used primarily with the "LSET" and "RSET" commands to place numeric data into the buffer associated with an open random file. The syntax for the "MKS\$" command is:

Compiler BASIC and Interpreter BASIC	
MKS\$(exp)	Function
exp	Is a numeric constant or expression of the desired type.

Since only strings may be stored in the buffer associated with an open random file, there exists a need to change numeric data into a string form. MKS\$ provides a way to change numeric data into a string. The string formed by "MKS\$" will always be four bytes in length, regardless of the actual value to be converted. The resultant string value obtained when performing an "MKS\$" command will be the compressed form of a number, contained in a four-byte string. After a numeric value has been changed into a four byte compressed string, it may then be placed into a buffer via the "LSET" and "RSET" commands. (Note: This is not the same as the "STR\$" command, as "STR\$" produces an ASCII string, not a compressed string representation of a number.) To convert a four-byte compressed representation of a number back to a numeric value, use the "CVS" command.

Suppose you have opened and fielded a random file, and wish to place a single precision value into the buffer. The fielded variable you are dealing with is "A4\$", and the value you wish to place in the part of the buffer pointed to by "A4\$" is contained in the variable "A4!". Then **LSET A4\$=MKS\$(A4!)** will cause a four-byte compressed string representation of the value stored in "A4!" to be written to the portion of the buffer pointed to by "A4\$". Note that the fielded length of the variable "A4\$" must be at least four bytes, and in most cases will be exactly four bytes.

&amp;O

&amp;O

This function indicates that the argument is an octal number rather than a decimal format number.

Compiler BASIC and Interpreter BASIC	
&Od0...d5 Octal number	Function

To represent a number in its octal format, you may use the characters "&O" as a prefix to the number. Interpreter BASIC also allows you to designate octal simply with the "&" prefix, thereby dropping the "O"; Compiler BASIC requires the "O".

One to six octal digits may follow the "&O" prefix. Octal digits consist of the numeric digits 0-7. The number represented using the "&O" prefix will always be taken as two's complement notation. The largest octal number which may be represented is &O177777.

For example, the assignments:

**A = B AND &O70**

and

**A = B AND 56**

are functionally equivalent. Other examples are:

**A=&O11**

A would be set equal to the decimal number 9.

**A=&170000**

A would be set equal to the decimal number -4096.

# OCT\$

# OCT\$

This DOS 6 Interpreter BASIC function returns in string form, the octal value of a number.

DOS 6 Interpreter BASIC	
<b>OCT\$(exp)</b>	<b>Function</b>
<b>exp</b>	Is any numeric expression.

In a manner similar to HEX\$ and BIN\$, OCT\$ will provide a character string representing the value of an expression which is expressed in octal digits: "0" through "7". For instance, after typing the following program and running it,

```
10 A$=OCT$(256)
20 PRINT A$
```

the display screen will show:

```
10 A$=OCT$(256)
20 PRINT A$
RUN
400
Ready
```

## ON exp

## ON exp

This statement allows your program to invoke conditional branching and subroutine calls.

### Compiler BASIC and Interpreter BASIC

**ON exp GOTO addrlist** Statement

**ON exp GOSUB addrlist** Statement

**exp** Designates the branch index of *addrlist*, and is in the range <0-255>.

**addrlist** Is a list of line numbers (or Compiler BASIC 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" or "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
```

## ON BREAK

## ON BREAK

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

### Compiler BASIC

**ON BREAK GOTO addr**

Statement

**addr**      Is either a LINE number 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.

Note that the function, "BRL", contains the line number where the BREAK occurred. This is similar to the "ERL" function used after an "ON ERROR GOTO" 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.

### 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...."
```

## ON ERROR

## ON ERROR

This statement is used for runtime program error control.

Compiler BASIC and Interpreter BASIC	
<b>ON ERROR GOTO addr</b>	Statement
<b>addr</b>	Is either a line number (or a Compiler BASIC label) which specifies the target of the branch.

During the execution of your BASIC program, if an error occurs, the program will cease. The response depends on whether it is an Interpreter BASIC or Compiler BASIC program. Compiler BASIC will normally print an error message,

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

and stop program execution when a runtime error is detected. An Interpreter BASIC program will stop and display one of the BASIC error messages. You can control the program halt by using an "ON ERROR GOTO" statement. If an "ON ERROR GOTO addr" is active at the time the program error is detected, program execution branches to the address specified by the "ON ERROR GOTO" statement on occurrence of a runtime error. Once active, the statement "ON ERROR GOTO 0" disables this feature and causes the visual error message previously mentioned.

To continue the program's execution, use the "RESUME" statement.

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

# OPEN

The OPEN command allows you to open random or sequential data files in order that input/output may occur between the computer and the given file. The general syntax for the "OPEN" command is:

## Compiler BASIC and Interpreter BASIC

**OPEN "type\$",bufnum,"filespec\$"[,reclen]    Statement**

<b>type\$</b>	Is the type of file access you wish to deal with (random or sequential). It may be represented as a string constant enclosed within quotes, or as a string expression. (See Type Tables)
<b>bufnum</b>	Is the number of the buffer (file control block) you wish to use to perform the disk I/O; this may be a numeric constant or a numeric expression, and must be an integer value within the range of 1 to the total number of active files declared when entering BASIC , (or with ALLOCATE).
<b>filespec\$</b>	Is the name of the disk file or device to access. It may be represented as a string constant or a string expression.
<b>reclen</b>	Is an optional expression in the range (1-255) designating the number of bytes in each record of the file to be opened. This pertains to <b>random files only</b> and should match the previous record length if the file already exists. If <i>reclen</i> is not used, it will default to 256. If <i>reclen</i> is specified as 0, it will be assumed to be 256. If BLK=OFF is specified when entering DOS 5 BASIC, <i>reclen</i> cannot be specified in an OPEN statement, and will default to 256.

Before a disk file can be manipulated it must first be OPENed. Also, before any compiler BASIC 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 the BASIC interpreter.

In order to write information to and retrieve information from a disk file, the file must be opened using the "OPEN" command. The "OPEN" command establishes the capability of reading from and writing to a disk file by creating a file control block (FCB). This FCB contains information needed by the computer, so that the computer may interact with the disk file. In addition, the "OPEN" command establishes a buffer which is used by the computer as a temporary storage place for information that will pass between the computer and the disk file.

There are different allowable file types, depending on the BASIC version; however, there are really only three fundamental types of files: Interpreter BASIC supports both *Random Record Access* and *Sequential Access*; Compiler BASIC supports these two plus *list-directed Extended*. In all cases, the designated file type string character may be upper or lower case.

Sequential files are file types that allow for accessing data in a specified sequence. That is to say, if you wish to retrieve the tenth piece of information in a file, you must read in the nine data items preceding the item in question before it may be accessed.

Random access files are file types that allow you to directly access any piece of information in a file, regardless of the physical location of the data within the file. Because of this nature, random access files are also known as *direct* files.

It is beyond the scope of this manual to discuss the techniques involved in creating and accessing information in random and sequential files. What will be provided for you here is the syntax needed to open all types of random and sequential files.

### IMPORTANT NOTE

It is strongly advised that no data file be in an open state at any given time using more than one buffer. BASIC will allow you to open the same file at the same time using more than one buffer; however, this practice may lead to the destruction of data files on the diskette in question!!



### Opening sequential files.

There are two basic modes available for use when dealing with sequential files; the input mode, and the output mode. The following list shows all of the different "OPEN" commands that may be issued when dealing with sequential files.

Sequential File Access - Type Table			
Type	Direction	File Status	BASIC
"I"	Input	Must exist	I5,I6,C
"O"	Output	Old or new	I5,I6,C
"OO"	Output	Must exist (old)	I5
"ON"	Output	Must not exist (new)	I5
"E"	Output, extend	Old or new	I5,I6,C
"EO"	Output, extend	Must exist (old)	I5
"EN"	Output, extend	Must not exist (new)	I5

Note: I5=DOS 5 BASIC, I6=DOS 6 BASIC, C=Compiler BASIC

With sequential access, a file is read ("I") or written ("O" or "OE"), basically a byte at a time, with INPUT# or PRINT#, respectively. BASIC 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 Compiler BASIC "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).

The input mode of sequential files allows you to input information from an existing file. No output to the file may be done if it has been opened for input. The file to be opened for input must exist, or the **OPEN "I"** command will return a File not found error. Once the file has been opened, information may be retrieved from it using the "INPUT#" and "LINEINPUT#" commands.

The output mode of sequential files allows you to output information to the file. No input from the file may be done if it has been opened for

output. Once the file has been opened, information may be written out to it using the "PRINT#" command. There are up to six types of output modes available for use with sequential files; these vary with the BASIC.

The **OPEN"O"** output mode, available for all BASICs, functions in the following manner. If the file opened does not exist, it will be created, and information will be written to the file starting at the first byte of the file. If the file opened does exist, any information previously stored in the file will be lost, as the new information to be placed in the file will be written over the existing information, starting at the first byte of the file.

The **OPEN"OO"** output mode, available for DOS 5 BASIC, functions in the following manner. If the file opened does not exist, a File not found error will be generated, and the file will not be created. If the file opened does exist, **OPEN"OO"** will function identically to **OPEN"O"** in the case where the file already exists.

The **OPEN"ON"** output mode, available for DOS 5 BASIC, functions in the following manner. If the file already exists, you will not be allowed to open the file, and the error File already exists will be generated. The existing file will not be altered in any way. If the file does not exist, it will be created, and information will be written to the file starting with the first byte of the file.

The **OPEN"E"** output mode, available for all BASIC's, functions in the following manner. If the file does not exist, **OPEN"E"** will function identically to **OPEN"O"**. If the file already exists, the file will be opened, and any information that will be written to the file will be appended to the end of the existing information. The file will be extended to include both the old and the new information.

The **OPEN"EO"** output mode, available for DOS 5 BASIC, functions in the following manner. If the file does not exist, a File not found error will be generated, and no file will be created. If the file already exists, the file will be opened, and any information that will be written to the file will be appended to the end of the existing information. The file will be extended to include both the old and the new information.

The **OPEN"EN"** mode, available for DOS 5 BASIC, functions identically to the **OPEN"ON"** output mode.

### Example - Opening sequential files

Suppose that you wished to open a sequential file named MYDATA/SEQ, using buffer number 1. The statement used to open the file for input would be as follows:

**OPEN "I",1,"MYDATA/SEQ"**

If you wished to open the same file for output using buffer number two, one of the following commands could be used, depending on whether or not you request that the file be new or old, and whether or not you wish to extend the file:

<b>OPEN "O",2,"MYDATA/SEQ"</b>	All BASIC's
<b>OPEN "OO",2,"MYDATA/SEQ"</b>	DOS 5 BASIC
<b>OPEN "ON",2,"MYDATA/SEQ"</b>	DOS 5 BASIC
<b>OPEN "E",2,"MYDATA/SEQ"</b>	All BASIC's
<b>OPEN "EO",2,"MYDATA/SEQ"</b>	DOS 5 BASIC
<b>OPEN "EN",2,"MYDATA/SEQ"</b>	DOS 5 BASIC

### Opening random files

Unlike sequential files, when dealing with a random file, you have the capability of reading from and writing to the file using only one "OPEN" command. The statements "PUT" and "GET" differentiate between writing to the file and reading from the file, respectively. There are four different types of "OPEN" statements that may be executed when opening a random file. They vary with the BASIC and are:

Random File Access - Type Table			
Type	Direction	File Status	BASIC
"D"	Input/Output	old or new	I6
"R"	Input/Output	old or new	All BASIC's
"RN"	Input/Output	Must not exist.	I5
"RO"	Input/Output	Must exist	I5
"X"	I/O List Directed	old or new	C
Note: I5=DOS 5 BASIC, I6=DOS 6 BASIC, C=Compiler BASIC			

The compiler BASIC "FIELD" statement for an "R"-mode file supports a field length of 256. The corresponding "OPEN" statement should not

include a *reclen* argument; the default is a record length of 256. This allows you to field a 256-byte record in a single string variable.

```
10  CLEAR 1000: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
SYSTEM "list test/dat (h) "
```

The **OPEN"R"** mode, available for all BASIC's, functions in the following manner. The file specified will be opened whether it exists or not, and will be created if it does not exist. After the file has been opened, the buffer used in the "OPEN" statement may be fielded using the "FIELD" statement, and records may then be retrieved from or placed into the file via the "PUT" and "GET" statements. Note that DOS 6 BASIC accepts the *type* as either "R" or "D".

The **OPEN"RN"** mode, available for DOS 5 BASIC, functions in the following manner. If the file already exists, you will not be allowed to open it. The file will remain untouched, and the error File already exists will occur. If the file does not exist, it will be created, and the **OPEN"RN"** command will function in the same manner as the **OPEN"R"** command.

The **OPEN"RO"** mode, available for DOS 5 BASIC, functions in the following manner. If the file does not exist, no file will be created, and the error File not found will occur. If the file does exist, **OPEN"RO"** will function in the same manner as **OPEN"R"**.

Compiler BASIC also supports a fairly powerful random access file mode, **OPEN"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",bufnum,"filename\$",reclen**

where "reclen" is the desired record length; "reclen" must include two bytes for each string variable specified in the XFIELD statement. 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. Its format allows either numeric or string variables in its list. Array variables are not allowed in the list.

### Example - Opening random files

Suppose you wish to open a random file named MYDATA/RND, using buffer number 3, with record lengths of 52 bytes. The following "OPEN" command may be used to open the file.

```
OPEN "R",3,"MYDATA/RND",52
```

All BASIC's

For more information on using both random and sequential files, refer to FIELD, XFIELD, GET, PUT, LSET, RSET, INPUT#, LINEINPUT#, and PRINT#.

Random access, specified by an "R" type in all BASIC's (or "D" in DOS 6 BASIC) in the "OPEN" statement, implies that file manipulation will be done discretely with any selected individual record in the file via the "GET" (get or read record) and "PUT" (put or write record) commands, which are described in detail elsewhere in this manual.

The OPEN statement also allows you to open logical devices for sequential I/O. For example, the printer device may be opened for output via a statement of the form,

```
OPEN "O",1,"*PR"
```

The standard disk-type PRINT# statement would then be used to "print" to the printer.

```
OPEN "O",1,"*DO"  
PRINT#1,"Data output to a file named '*DO'"  
PRINT#1,"This is a continuation of output..."  
CLOSE 1
```

## OPTION BASE

## OPTION BASE

This DOS 6 statement establishes the minimum value for an array subscript as zero or one. It's syntax is:

### DOS 6 Interpreter BASIC

**OPTION BASE** *intvar*

Statement 6

**intvar**      Is the value of the base, 0 or 1.

BASIC arrays are normally established so that the first element is referenced with an index of zero; i.e. `ARRAY(0)`. Without use of any `DIMENSION` statement, arrays default to a maximum element index of 10; thus, undimensioned arrays can store eleven elements. The "DIM" statement is used to declare the specific maximum element of an array. But since the first element of an array is referenced as zero, an array actually has one more element than the magnitude of the dimension.

Some programmers prefer to avoid using index zero for an array element; but this will waste the space of one element for each array used. The "OPTION BASE" statement allows you to tell BASIC that you will not be using index 0; thus, BASIC avoids reserving memory space for the zeroth element. This statement must precede any "DIM" statement used to declare the bounds of an array.

Note: Do not specify **OPTION BASE 1** if you are going to use the **BSORT** array sorting utility available separately.

# OUT

# OUT

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

Compiler BASIC and Interpreter BASIC	
OUT portnum,value	Statement
<b>portnum</b>	Is a numeric expression which evaluates to the range <0 to 255>, specifying a CPU port number.
<b>value</b>	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.

### Programmer's Note:

Under Interpreter BASIC, the value of *portnum* can actually range from 0 through 32767; the actual value used for the port determination will be "*portnum* modulo 256". Thus, a *portnum* of 256 designates port 0; a *portnum* of 257 designates port 1. However, the high-order value will be presented to the high-order address lines of the CPU address bus. This may be useful in addressing internal and external ports of the 64180/Z180 processor which requires a high-order zero value to reference internal ports and a high-order non-zero value to reference external ports.

## **PAGELEN**

## **PAGELEN**

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

### **Compiler BASIC**

**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. Its use is similar to using the DOS FORMS filter with "Page=exp"; however, Compiler BASIC's paging control is strictly internal to BASIC.

For additional printer paging statements, see "LINESPAGE", "LMARGIN", "PZONE", and "RMARGIN".



# PAINT

# PAINT

This Compiler BASIC statement is used to fill in a bounded shape.

## Compiler BASIC

**PAINT(x,y)[,color]**

Statement C

<b>x,y</b>	Is the coordinate of a point interior to the bounded shape. 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.
<b>color</b>	Is the color used to fill the shape <0,1>; where black = 0 and white = 1. If <i>color</i> 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, fill in the interior of the triangle, then pause awaiting a key entry.

```
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$=WINKEY$
```

# PEEK

# PEEK

This function obtains the byte stored at a memory address.

## Compiler BASIC and Interpreter BASIC

**PEEK(exp16)**

Function

**exp16** Represents a memory address in the range  
<0 to 32767> and <-32768 to -1> (i.e. 0 to 65535).

PEEK is a means to “look” 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.

# PLOT

# PLOT

This Compiler BASIC statement is used to plot a line of pixels.

Compiler BASIC	
PLOT flag,x1,y1 TO x2,y2	Statement
<b>x1,y1</b>	Specifies the coordinate point of one line endpoint.
<b>x2,y2</b>	Specifies the coordinate of the other line endpoint.
<b>flag</b>	Single character 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 complement a line on the screen. The coordinates **x1,y1** and **x2,y2** range as follows: **X1** and **x2** are numeric expressions which evaluate to the range <0 - 127> for 64-column screens and <0 - 159> for 80-column screens. **Y1** and **y2** are numeric expressions which evaluate to the range <0 - 47> for 16-row screens and <0 - 71> for 24-row screens. 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 *flag* value can also take on any of the three values: "SB", "RB", or "CB". These flag values stand for "set box", "reset box", and "complement box" respectively. These arguments direct **PLOT** to plot a box using the two coordinate pairs as the northwest and southeast corners of a rectangle.

Note that for the "PLOT" graphics statement, a coordinate value greater than the legal range will be truncated to the maximum permissible value. Note also that only the low-order byte will be tested for validity.

The following program makes an interesting fan-line 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
```

# POINT

# POINT

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

<b>Compiler BASIC and DOS 5 Interpreter BASIC</b>	
<b>POINT(x,y)</b>	<b>Function</b>
<b>x,y</b>	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 (**TRUE**) if the point is **SET**, 0 (**FALSE**) otherwise. Because of these response values, "**POINT**" is typically used as the expression under test in an "**IF**" statement.

# POKE

# POKE

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

Compiler BASIC and Interpreter BASIC	
<b>POKE exp16,exp8</b>	Statement
<b>exp16</b>	Specifies a memory address in the range <0 to 32767> and <-32768 to -1> (i.e. 0 to 65535).
<b>exp8</b>	Is a numeric expression which evaluates to the range <0 to 255>.

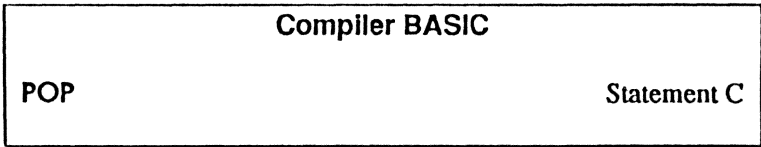
POKE allows direct modification of any single byte RAM location in memory. "POKE" is the complement of "PEEK".

Compiler BASIC also provides "WPOKE" to allow direct modification of any two-byte (word) 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.

# POP

# POP

This Compiler BASIC statement is used to escape from a GOSUBed subroutine.



**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.

# POS

# POS

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

<b>Compiler BASIC and Interpreter BASIC</b>	
<b>POS(dummy exp)</b>	<b>Function</b>
<b>dummy exp</b> Is a required expression which is ignored.	

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

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

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



## POSFIL

## POSFIL

“POSFIL” is a Compiler BASIC statement which allows you to position a sequential input/output file pointer for subsequent Input/Output operations.

Compiler BASIC	
POSFIL(#bufnum,recnum,offset)	Statement
<b>bufnum</b>	Is a file control block buffer number, <1-15> which corresponds to an open file.
<b>recnum</b>	Is the disk file's 256-byte record number.
<b>offset</b>	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 five records).

# PRINT

# PRINT

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

## Compiler BASIC and Interpreter BASIC

**PRINT [#numexp,][@pos,] [item][,][;]  
[TAB(exp)] ...**                      Statement  
Statement

**numexp**    Is a numeric expression within the range <-3 to 15>;  
1 thru 15, send to disk file. For compiler BASIC,  
-3, send to PRINTER; 0, send to VIDEO display.

**pos**            Is a numeric expression between 0 and 1023 [0-1919  
for DOS 6] specifying a new cursor relative position.  
For DOS 6 BASIC, *pos* can be entered  
as (row,column).

**item**            Is a "string literal" or a numeric / string expression;  
a list of items may be provided.

**,;**            Are delimiters.

**PRINT** is the facility used to display data on the video screen. **PRINT#**, a version of "PRINT", is also used to write data to a sequential disk file (type "O" or "E"); "PRINT#" is discussed in detail under its own heading. All "LPRINT" statements are equivalent to "PRINT" statements, but are directed to the line printer rather than the video screen.

The data to be displayed may be a single item or a list of items. There are four options for terminating a variable in the item list. A comma delimiter or equivalently TAB(255) tabs the cursor to the next screen zone. These zones are at intervals of 16 character positions (0, 16, 32, 48, 64). A semi-colon delimiter retains the print position immediately following the output of *item*; however, a trailing blank space follows all numbers. By not specifying a terminator, the next "PRINT" will occur on the next print line. You can **tab** to a particular column by specifying a "TAB(exp)" with *exp* equal to the column position. For example:

```
10 CLS:I=15.8
20 PRINT "012345678901234567890123456789"
30 PRINT I;TAB(26);I
RUN

012345678901234567890123456789
15.8                               15.8
```

When execution of a program in BASIC is ended, and you are returned back to the BASIC Ready prompt, BASIC turns the cursor on. If you do a "PRINT CHR\$(15);" from the BASIC Ready prompt, it may look like nothing happens. Indeed the net result is that you are back where you started. But put exactly the same statement inside a program, followed by a statement that should present a cursor, "INPUT A\$" for example, and the cursor is gone! Then comes the end of the program and the BASIC Ready prompt, complete with cursor. This can affect video that is routed to printers if they happen to respond to this code. If you BREAK out of a program, the cursor will be turned back on at Ready and will still be on when you CONTINUE.

All "PRINT" statements used in Interpreter BASIC programs should compile and function with equivalence with no modifications necessary under Compiler BASIC.

The default screen or printer TAB positions (i.e. every 16 columns) 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 a string expression under Compiler BASIC. Compiled and interpreted BASIC "PRINT USING" statements usually produce the same output; however, under Compiler BASIC, the target string of a "USING" may be manipulated with the string functions just like any other string.

## PRINT#

## PRINT#

The **PRINT#** command allows you to output data to a sequential file. The syntax is:

Compiler BASIC and Interpreter BASIC	
<b>PRINT#bufnum, item list</b>	Statement
<b>bufnum</b>	Is the buffer used to open the file. It may be expressed as a numeric constant or a numeric expression.
<b>item list</b>	Is a list of constants and/or expressions that contains the data you wish to output to the file.

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

Numeric constants, numeric expressions, string constants and string expressions may all be contained within the item list. If more than one value is to be output to the file using a single "**PRINT#**" statement, these values must be separated by some type of delimiter. The uses of delimiters in a **PRINT#** command will be explained throughout this section.

The "**PRINT#**" command is used in conjunction with any type of **OPEN"O"** or **OPEN"E"** command. After a file has been opened, data may be output to the file via the "**PRINT#**" command. Once a file has been created using the **OPEN"O"** or **OPEN"E"** and subsequent "**PRINT#**" commands and then closed, the information in the file may be accessed using the **OPEN"I"** and "**INPUT#**" or "**LINEINPUT#**" commands.

In most cases, data written to a sequential file is stored in ASCII format. For numeric data, a sign byte will always precede the numeric information. If the value is positive, the sign byte will be represented by a space. A trailing space will always follow the ASCII representation of the

value. Keeping the above in mind, the minimum amount of bytes required to store a numeric value in a sequential file is three (the sign byte, a digit, and the trailing space).

For string data, all characters included in the string value will be written to the file, and no preceding or trailing characters will be written to the file. Special considerations do need to be taken into account when writing string values to a sequential file, as there are some peculiarities involved with the "INPUT#" command when trying to access string information stored in a sequential file. These special cases will be pointed out throughout this section.

The "PRINT#" command resembles the "PRINT" command in many ways with respect to how information is physically written to the file. Some of the punctuation used in the "PRINT#" command will cause data to be written to the file in much the same way that this punctuation causes data to be printed to the screen using the "PRINT" command.

Punctuation is very important when using the "PRINT#" command. The following will describe the punctuation which is allowed with the "PRINT#" command, and the effects of using different punctuation.

### Use of punctuation with the PRINT# command.

Different types of punctuation used to separate values to be output in a "PRINT#" statement will cause the data to be physically written to the file in different ways. The following list shows the punctuation required to separate values contained in a "PRINT#" statement.

PRINT# Punctuation List	
,	comma
;	semicolon
","	explicit comma

When separating output data contained in a "PRINT#" statement, you may use either a comma or a semicolon. A semicolon will cause the next piece of information to be written directly after the preceding data. A comma will cause the next piece of information to be written at the next available "tab" position in the file. Tab positions will be denoted by 16-byte blocks, starting from the last occurrence of a carriage return (ODH) in the file.

In some cases, the explicit comma is used after string information has been written to the disk, to demarcate the end of the string value from the beginning of the next piece of information to be written out.

The following examples will illustrate the methods used to write data to a sequential file, as well as the occurrences that will result when this data is to be retrieved.

### **Example 1 - Writing numeric data to a sequential file.**

Suppose you wish to write two numeric values out to a sequential file, using one "PRINT#" command. The file you wish to write these values out to is named DATA1/SEQ, and has been opened using buffer number 2. The variables you wish to write out to the file are "A%", which has been assigned a value of 362, and "B!", which has been assigned a value of -2618.7. The following "PRINT#" command may be used to write these values out to the file:

**PRINT#2,A%;B!**

The above statement will cause the values 362 and -2618.7 to be written to the file in ASCII format. The image produced on the disk by this "PRINT#" statement is shown below. (Note that throughout the rest of this section, the image produced by the example "PRINT#" statements will always follow the "PRINT#" statement. The image shown will be similar to the **DOS LIST (H)** command; each ASCII character will be displayed with its corresponding hex value shown below the character.)

3	6	2		-	2	6	1	8	.	7		
20	33	36	32	20	2D	32	36	31	38	2E	37	20 0D

Note the sign byte preceding each value, and the trailing space following each value. Also note that the last byte written to the file is a carriage return (0DH). A carriage return will always be written to the file after the last item listed in a "PRINT#" statement.

Realize that a *semicolon* was used to separate the variables "A%" and "B!" in the "PRINT#" command. A *comma* could have been used instead; however, the image of the data on the disk would have changed to the following if a comma would have been used instead of a semicolon.

3	6	2														
20	33	36	32	20	20	20	20	20	20	20	20	20	20	20	20	
-	2	6	1	8	.	7										
2D	32	36	31	38	2E	37	20	0D								

Notice the series of spaces following the number 362. These will be written to the disk as a result of a *comma* being used to separate the variables "A%" and "B!". As was noted earlier, when using a *comma* to separate variables in a "PRINT#" statement, the value following the comma will be written to the next tab position (the beginning byte of the next block of 16 bytes). As depicted in the above displays, much disk space will be wasted in writing to sequential files if the values in a "PRINT#" statement are separated by commas instead of semicolons.

### Example 2 - Writing string data to a sequential file.

Suppose you wish to write three string values out to a sequential file, using one "PRINT#" command. The file is named "DATA2/SEQ", and has been opened using buffer number 1. The variables you wish to write out to the file are "A\$" (which has been assigned the value "AMBER"), "B\$" (which has been assigned the value "BROWN"), and the string constant "GRAY". The following "PRINT#" command may be used to write these values out to the file:

**PRINT#1,A\$;",";B\$;",";"GRAY"**

The above statement will cause the values "AMBER", "BROWN" and "GRAY" to be written to the file. The image produced on the disk by this PRINT# statement is shown below.

A	M	B	E	R	,	B	R	O	W	N	,	G	R	A	Y
41	4D	42	45	52	2C	42	52	4F	57	4E	2C	47	52	41	59 0D

There are many things to be noted in this example. The most prominent of these is the use of the explicit comma (","). You will note from the above display that along with the string values, commas were also written out to the file (since they were enclosed within quotes as part of the list of values to be written out). In most applications dealing with writing strings out to sequential files, you will need to incorporate the explicit comma within the list of values to be printed out by the "PRINT#". The reason behind

this stems from the way "INPUT#" deals with retrieving information from a sequential file.

Before continuing with more examples on the use of "PRINT#", a brief discussion of using "INPUT#" with files created by "PRINT#" is in order.

### How INPUT# ties together with PRINT#

As shown throughout this section, the punctuation used in the "PRINT#" command is very important, and determines the manner in which "INPUT#" will access this information. "INPUT#" deals with retrieving numeric data in a different fashion than it does with string data.

When "INPUT#" requests the input of a numeric variable, it will begin reading from the last accessed byte in the file. Any leading spaces that are encountered will be ignored. Once "INPUT#" finds a non-space character, it will read until it encounters either a space or a delimiter, and the value assigned to the variable will be determined by performing a VAL function on the characters read in. This is to say that any characters may be input into a numeric variable, and the inputting of string values into a numeric variable will not cause a Type mismatch error.

When "INPUT#" requests the input of a string variable, it will begin reading from the last accessed byte in the file, and proceed until it finds a non-space character. Once it finds a non-space character, it will read until it encounters a delimiter, and the value assigned to the variable will be all characters read in from the first non-space character to the delimiter. Note from the above description that any "leading" spaces which are present in the data file for the data element in question will be ignored by "INPUT#", and the value assigned to the string will never have leading spaces.

In all cases, when "INPUT#" requests an input of a variable, the input will be terminated when a delimiter character is read in. For numeric inputs, delimiters can be represented by either a space, a comma, or a carriage return (ODH). In most cases, a comma should not be used as the delimiter for a numeric input.

For string inputs, a delimiter can be represented by either a comma or a carriage return. Realize that for any input of a variable, if the number of



characters read in will exceed 255, the input of the variable will terminate after the 255th character has been accessed.

One point to note is that in most cases, two delimiter characters should not appear together in a sequential file. This occurrence will cause unpredictable results when trying to input information from the file.

From the above paragraphs, it can be seen that in any one physical "PRINT#" statement, if values are to be written out following a string value, they must be separated from the string value by use of the explicit comma. The general format which is recommended to perform such a data write is as follows:

**PRINT#b,...;string value;",";next value;...**

### **Example 3 - Writing numeric and string data to a file.**

Suppose you wish to write several string and numeric values out to a sequential file using the same "PRINT#" statement. The file you wish to write these values out to is named DATA3/SEQ, and has been opened using buffer number 2. The string values you wish to write out are contained in the variables "A\$" (which has been assigned the value "ANN"), B\$ (which has been assigned the value "BETTY") and "C\$" (which has been assigned the value "CAROL"). The numeric values you wish to write out are contained in the variables "A%" (which has been assigned a value of "20"), "B%" (which has been assigned a value of "32"), and "C%" (which has been assigned a value of "23"). The following will show a "PRINT#" statement which may be used to write these values out to the file, and the associated image that will be written to the disk as a result of performing the PRINT#.

PRINT#2,A%;A\$;",";B%;B\$;",";C%;C\$																	
2	0			A	N	N	,			3	2			B	E	T	T
20	32	30	20	41	4E	4E	2C	20	33	32	20	42	45	54	54		
Y	,		2	3			C	A	R	O	L						
59	2C	20	32	33	20	43	41	52	4F	4C	0D						

Please note from the above example that no explicit comma needs to follow numeric data. Also note that since "C\$" is the last variable to be written out in this "PRINT#" command, no explicit comma is needed after it, as a carriage return will always be written out to the file after the last

variable in a "PRINT#" command. This carriage return will serve as the delimiter for subsequent "PRINT#" commands.

This concludes our discussion of the "PRINT#" command. It is recommended that you create test files in order to explore the results of various "PRINT#" statements. After sequential files have been created, they may be examined by use of the DOS LIST **filespec (H)** command. For further information, see "OPEN", "INPUT#", and "LINEINPUT#".

## PRINT USING

## PRINT USING

The **PRINT USING** command (and its associated “**PRINT# USING**”) will allow you to output data using a specified format. The syntax for the “**PRINT USING**” command is:

### Compiler BASIC and Interpreter BASIC

**[L]PRINT[#bufnum] USING format\$;explist      Statement**

**bufnum**      When used, is the buffer used to open the file. **PRINT** output is then directed to the sequential file identified by *bufnum*.

**format\$**      Is the control string format you wish to use to output the list of values. It may be represented as either a string constant or a string expression.

**explist**      Is the list of constants and/or expressions to output.

The **PRINT USING** command will allow you to output data to the video screen in the format specified by the format string. Similarly, **LPRINT USING** directs the same output to the line printer. The **PRINT# USING** command will allow you to output the data to a sequential file in the format specified by the format string. Any format string which is allowable in the “**PRINT USING**” command will also be allowable in the “**PRINT# USING**” command, and will function in an identical manner. For more information on the specifics involved in writing information out to a sequential file, see “**PRINT#**”.

“**USING**” is actually a string expression under Compiler BASIC. Compiled and interpreted BASIC “**PRINT USING**” statements usually produce the same output; however, under Compiler BASIC, the target string of a “**USING**” may be manipulated, exclusive of any “**PRINT**” statement, with the string functions just like any other string.

The complete field specifier list for "USING" is as follows:

Field Specifier List		
Numeric Formats		
<u>Spec.</u>	<u>Purpose / definition</u>	<u>Example</u>
#	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 a comma before every 3rd digit left of dp.	
+	Print leading/trailing sign (either + or -)	+###.## ###.##+
-	Print trailing sign if negative, or SPACE if positive	####.##-
**	Fill unused digits with asterisks instead of blanks; adds 2 field positions	**###.##
\$\$	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	###.#### I6,C6
↑ ↑ ↑ ↑		I5,C5
_ (underline)	Print next character as a literal character.	I6
Note: I5=DOS 5 Interpreter; I6=DOS 6 Interpreter; C=Compiler		

String Formats		
Spec.	Description	Example
!	First character of string expression	"!";"ABC" = "A"
%blanks%	Include 2+# of blanks	"% %"; I5,C
\blanks\	length substring of string expression	"ABCDE" = I6 "ABC"
&	Print the string without modifications.	I6

Note: I5=DOS 5 Interpreter; I6=DOS 6 Interpreter; C=Compiler

Note that when using the "\*\*\*", "\$\$", or "\*\*\*\$" specifier, two additional field positions will be specified.

## Examples:

Column Positions.....	=	"123456789"
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"	=	"A B"
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--"

Suppose you wish to write three numeric values out to a sequential file. The name of the file is DATA/SEQ, and it has been opened using buffer

number 1. The values you wish to write out are contained in the variables "A%" (which has been assigned a value of 25), "B!" (which has been assigned a value of 13.73), and "C%" (which has been assigned a value of -17). The format string you wish to use has been assigned to the variable "A\$", and has the value:

```
### #####.### #####
```

The following will show a "PRINT# USING" command that may be used to write out the above values, and the disk image created by the "PRINT# USING" command.

```
PRINT#1,USINGA$;A%,B!,C%
```

```

  2 5          1 3 . 7 3 0          - 1 7
20 32 35 20 20 20 20 20 20 31 33 2E 37 33 30 20 20 2D 31 37 0D

```

Note from the above example that the image created on disk conforms to the format string specified. Unlike the "PRINT#" command, the use of delimiters to separate the values to be printed out is arbitrary. That is to say, there is no difference in using a comma as a delimiter as opposed to a semicolon.

Note that Compiler BASIC allows you to also direct the output of a "PRINT#bufnum" command to either the video screen or your printer by specifying *bufnum* as 0 or -3 respectively. Expressing the *bufnum* as a variable permits you to designate the output device at runtime. Thus, the command **PRINT#-3,"This is a test"** will print the text string on your printer. 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

```

## PUT

## PUT

PUT is used to write information from a record buffer to a specified record of a random file. The information that is to be written out to the file must have been placed into the buffer that was used to open the file prior to being written out to the file. The syntax for the "PUT" command is:

Compiler BASIC and Interpreter BASIC	
PUT bufnum[,recnum]	Statement I
PUT bufnum,recnum	Statement C
<b>bufnum</b>	Is file control block buffer number, 1-15, used to open the file
<b>recnum</b>	Is the record number to write; it is required for compiler BASIC but optional for interpreter BASIC.

PUT and GET are the two type "R" (or DOS 6 Interpreter type "D") 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 for Compiler BASIC. Under Interpreter BASIC, if the record number, *recnum*, is not specified, BASIC will first increment the current record number by one, after which it will perform a "PUT" of the current record number. If no current record number has been established, the computer will perform a "PUT" of record number one, and the current record number will be set equal to one.

Suppose you wish to output data to a random file. The file you wish to perform the output to has the name FILE/RND, and has been fielded using buffer number 2. The record you wish to write out to the file is record number 23. Assume also that all of the values you wish to write out to the file have been placed into the buffer using the proper "LSET" and "RSET" commands. One of the following "PUT" commands may be used to write the information to the 23rd record of the file.

### PUT 2,23

**N%=1:N1%=30:PUT N%+1,N1%-7**

After executing one of the above statements, the information stored in the buffer associated with the file, FILE/RND, will be written out to the disk, and will be placed in the file as representing the 23rd record in the file. Once this information has been placed into the file, it may be retrieved using the "GET" command.

For more information on using "PUT", see "OPEN", "FIELD", "XFIELD", "LSET" and "RSET".



## PZONE

## PZONE

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

Compiler BASIC	
<b>PZONE(pos 1,...,pos n)</b>	Statement
<b>PZONE(*)</b>	
<b>pos</b>	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. For example, consider the following illustrative program and its associated screen display.

```
PRINT"012345678901234567890123456789012345
FOR I = 1 TO 3
PRINT I,
NEXT
PRINT:PZONE(10,15,30)
FOR I = 1 TO 4
PRINT I,
NEXT
```

012345678901234567890123456789012345
<div style="display: flex; justify-content: space-between; width: 100%;"> <span>1</span> <span>2</span> <span>3</span> </div>
<div style="display: flex; justify-content: space-between; width: 100%;"> <span>1</span> <span>2</span> <span>3</span> <span>4</span> </div>

For additional printer paging statements, see "LINESPAGE", "LMARGIN", "PAGELEN", and "RMARGIN".

# RANDOM

# RANDOM

This statement seeds the random number generator.

Compiler BASIC and interpreter BASIC	
<b>RANDOM [exp]</b>	Statement
<b>exp</b>	Is an optional integer expression in the range <0-255> used to seed the generator.

RANDOM reseeds the “random” number generator to assure a high probability of a non-repeating “random” sequence of numbers.

Compiler BASIC 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 Interpreter BASIC RND results.

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.

## RDGOTO

## RDGOTO

This compiler BASIC statement allows you to reset the DATA list pointer.

Compiler BASIC	
<b>RDGOTO addr</b>	Statement
<b>addr</b>	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** 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
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
```

# READ

# READ

This statement allows you to declare and read a list of data items.

## Compiler BASIC and Interpreter BASIC

**READ** var1 [,var 2,...,var n] Statement

**var** Is either a numeric or string variable or array element.

**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 eight-byte string. **READ A**, using the same item, stores into “**A**” the binary equivalent of the converted string 1.618033.

Initially, the first data item read, unless the data pointer is changed by a “**RDGOTO**” or “**RESTORE**” statement, will be the first data item in the first **DATA** statement in the program. “**RESTORE**” and “**RDGOTO**” provide ways to point at the desired data list. Note the following example program:

```
10 READ TITLE$:PRINT TITLE$:PRINT:READ N
20 FOR X=1 TO N:READ A:?,A,:NEXT
30 END
40 DATA The first 8 Fibonacci numbers in order
50 DATA 8, 1,1,2,3,5,8,13,21
```

# REM

# REM

This statement is used to enter a remark into your source program.

Compiler BASIC and Interpreter BASIC	
--------------------------------------	--

REM info or ' info	Statement
--------------------	-----------

REM, or the apostrophe character, signals the compiler to ignore the rest of the source line. Nothing included on the line after a “REMark” statement is included in the compiled program.

Compiler BASIC requires that a colon precede the apostrophe if the remark is part of a multiple statement. For example,

```
20  %LOOP0(10,0,1):  'Note 10 > 0
```

requires the “:” to make the remark form a new statement.

## REPEAT - UNTIL

## REPEAT - UNTIL

These Compiler BASIC statements implement the typical REPEAT-UNTIL loop construct.

### Compiler BASIC

**REPEAT**

Statement C

statements

**UNTIL exp**

Statement C

**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. Compiler BASIC 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). Program execution resumes at the statement following "UNTIL exp" if *exp* is equal to zero (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:

- The initial loop index variable value,
- the top limit, and
- 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
```



# RESET

# RESET

This statement is used to turn off a pixel.

## Compiler BASIC and DOS 5 Interpreter BASIC

RESET(x,y)	Statement
<b>x</b>	Is a numeric expression which evaluates to the range <0 - 127> for 64-column screens and <0 - 159> for 80-column screens.
<b>y</b>	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 BASIC command that unconditionally turns on any selected block graphic's pixel on the video screen. The "RESET" command turns a pixel "OFF". The Compiler BASIC "COMPL" command complements a selected graphic's pixel, turning it "ON" if it is "OFF" and vice versa. A function, "POINT(x,y)", is also related to the pixel graphics commands.

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.

# RESTORE

# RESTORE

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

Compiler BASIC and Interpreter BASIC	
RESTORE	Statement C
RESTORE [line]	Statement I

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. The RESTORE command of Interpreter BASIC allows you to specify a line number; the data pointer will be reset to the specified line, and any subsequent READ statements will start from that line. This command must be the first statement in a program line.

Compiler BASIC's "RDGOTO" can also be used to reposition the list pointer to any line or labeled location of the data list. **RDGOTO line** is equivalent to **RESTORE line**.

### Example Program:

```
5    RESTORE 80
10   READ TT$:PRINT TT$:PRINT:READ N
20   FOR X=1 TO N:READ A: ?A, :NEXT
30   END
50   DATA The first 8 Fibonacci numbers in order
60   DATA 8, 1,1,2,3,5,8,13,21
80   DATA The first NINE prime numbers in
sequential order
90   DATA 9, 2,3,5,7,11,13,17,19,23
```

# RESUME

# RESUME

This statement performs an unconditional program branch. It is used primarily in an error-trapping routine.

## Compiler BASIC and Interpreter BASIC

<b>RESUME <i>addr</i></b>	Statement
<b>RESUME [<i>line</i>]</b>	Statement I
<b>RESUME NEXT</b>	Statement I

<b><i>addr</i></b>	Is a line number or a label.
<b><i>line</i></b>	Is a BASIC program line number.
<b>NEXT</b>	Implies the Interpreter BASIC line number following the statement where the error occurred.

**RESUME** is used in an "ON ERROR GOTO" error handling routine and provides the means to continue running the program after the error has been handled.

Under Interpreter BASIC, omitting the *line* operand is equivalent to "RESUME 0". This will cause a return to the statement in which the error occurred. A "RESUME *line*" will cause a return to the statement with the specified line number. Finally, A "RESUME NEXT" returns to the line number following the statement where the error occurred.

Compiler BASIC does not support a "RESUME NEXT". You can "RESUME *addr*", where *addr* is at a highest program level (i.e. not contained in a GOSUB'd subroutine or program loop). "RESUME" resets the program stack pointer to its initial value, normalizes the program code pointer, then performs a "GOTO".

# RETURN

# RETURN

This statement is used to return from a GOSUBed subroutine.

Compiler BASIC and Interpreter BASIC	
RETURN	Statement

GOSUB is the standard BASIC command to call a subroutine. Nested "GOSUB" 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 Compiler BASIC "POP" command documented elsewhere.

```

10  DIM A(10),B(10):' Compiler 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"SORTA":' Could be GOSUB 110
40  GOSUB"PRINTA":'Could be GOSUB 140
50  GOSUB"SORTB":' Could be GOSUB 130
60  GOSUB"PRINTB":'Could be GOSUB 110
70  END
80  '
100 "SORTA":'   Alternatively: JNAME"SORT A"
110 SCLEAR:KEY A(0):TAG B(0):SORT 11:RETURN
120 "SORTB"
130 SCLEAR:KEY B(0):TAG A(0):SORT 11:RETURN
140 "PRINTA"
150 FOR X=0 TO 10: PRINT A(X),B(X):NEXT:RETURN
160 "PRINTB"
170 FOR X=0 TO 10: 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.

# **RIGHT**

# **RIGHT**

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

<b>Compiler BASIC</b>	
<b>RIGHT</b>	<b>Statement</b>

**RIGHT** scrolls the entire screen right by one character, clearing the leftmost (0th) screen column.

## RIGHT\$

## RIGHT\$

This function extracts the right-hand sub-string of a string.

Compiler and Interpreter BASIC		
<b>RIGHT\$(exp\$,exp1)</b>		Function
<b>exp\$</b>	Is any string expression.	
<b>exp1</b>	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**".

## RMARGIN

## RMARGIN

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

### Compiler BASIC

**RMARGIN = exp**

Statement

**exp**      Is a numeric expression which evaluates to the range <0-255>; 0 implies no margin checking.

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*.

The **RMARGIN** statement allows an "**RMARGIN=0**" which results in the total suppression of left-margin indents, "**PAGELEN**" checking, as well as right-margin checking. This provides the capability of generating printer controls without Compiler BASIC adding any page formatting.

For additional printer paging statements, see "**LINESPAGE**", "**LMARGIN**", "**PAGELEN**", and "**PZONE**".



## **RND**

## **RND**

This function obtains a random number.

Compiler and Interpreter BASIC	
<b>RND(exp)</b>	Function
<b>exp</b>	Is a numeric expression.

**RND** returns a pseudo-random number between 0 and 0.999999 if *exp* is equal to zero; 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).

Interpreter BASIC performs its random number generator calculations in single precision.

All calculations in Compiler BASIC 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.

# ROT

# ROT

This compiler BASIC statement is used to establish a rotation for the "DRAW" statement.

Compiler BASIC	
<b>ROT = exp8</b>	Statement
<b>exp8</b>	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. The following program draws a figure, using "DRAW", then rotates the figure about the plot origin.

```
10 DEFINT F:CLS:DIM FIGURE1(110)
20 Y=0:FOR X=0 TO 250 STEP 10
40 FIGURE1(Y)=X*6:'Line length = 6*X/256 units
50 FIGURE1(Y+1)=X+256:'Rot = X, entry code = 1
55 Y=Y+2
60 NEXT:'      Continue until figure completed
70 FIGURE1(Y+1)=0:'0 entry to terminate list
75 ' Draw it!
    FOR I = 0 TO 255 STEP 16:ROT=I:'Rotate figure
80 DRAW SET @64,23 USING FIGURE1(0)
    DRAW RESET @64,23 USING FIGURE1(0)
    NEXT I:A$=WINKEY$
```

Drawing begins at location (64,23) and the object is SET on the screen as per the DRAW flag "SET". To demonstrate the rotation available with "ROT", the figure is reset immediately after being drawn.

# ROW

# ROW

This function obtains the current row position of the cursor.

Compiler BASIC and Interpreter BASIC	
ROW(dummy exp)	Function
dummy exp Is a required expression which is ignored.	

ROW returns the row of the cursor; equal to "INT((CURLOC)/(number of columns)". For example,

```
10 PRINT@170,"H1."
20 A = ROW(0)
```

assigns a 3 to "A".

## RSET

## RSET

The RSET command will allow you to place information into the buffer associated with a disk file, prior to writing this information out to the disk. It is used primarily in conjunction with random files.

### Compiler BASIC and Interpreter BASIC

**RSET var\$ = exp\$**

Statement

**var\$** Is the variable used in the field statement that points to the buffer location where the data is to be placed.

**exp\$** Is the information to add; it must be a string constant or string expression.

The RSET command functions identically to the "LSET" command, with the following exception. Rather than the information being placed into the buffer left-justified, "RSET" will place the information into the buffer **right justified**. If the length of the string to be placed into the buffer is less than the fielded length of the particular slot of the buffer, spaces will be inserted in front of the string in the buffer to make the string in the buffer the same length as specified in the field statement.

If the length of the string to be "RSET" into the buffer is greater than the fielded length, the right most part of the string will be placed in the buffer, and any characters to the left of the total allocated space will be truncated (i.e. the information is "right justified").

The main difference between "MID\$" and "LSET" or "RSET" is that the latter commands fill the remaining characters in the affected string with blanks, or CHR\$(32)'s. Note that "LSET" and "RSET" commands work on any string variable, not just FIELDed string variables.

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

```
RSET A$="12345678912"   Now A$="2345678912"  
RSET A$="HELLO"         A$="  HELLO"
```

## RUN

## RUN

Compiler BASIC's "RUN" will load a "/CMD" type program from disk and then invoke it. Interpreter BASIC's "RUN" command will allow you to load a BASIC program stored on disk into the computer's memory, and immediately begin execution of that program.

### Compiler BASIC and Interpreter BASIC

RUN ["filespec\$"]   [line]	Statement C
RUN ["filespec\$"] [R]	Statement I6
RUN ["filespec\$"] [,R V] [,line]	Statement I5

**filespec\$** Designates the file to run.

**R** Is used with Interpreter BASIC to leave open all currently open files prior to running *filespec\$*.

**V** Is used to leave open all currently open files prior to running *filespec\$* as well as retain all variables [similar to "CHAIN *filespec\$,ALL*"].

**line** Will re-run the current program at line, *line*.

### Interpreter BASIC RUN

The RUN command may be issued from the BASIC Ready prompt to load and execute a program, or may be used from within a BASIC program to perform a chaining of programs. If the RUN command is given with a *filespec\$*, any program which is currently resident in memory will be overwritten, and the program specified in the RUN command will be loaded and executed. *Filespec\$* is the name of the program that you wish to be loaded and executed, and may be represented by any valid DOS filespec; it may be either a string constant or a string expression. If *filespec\$* is not included, the program currently in memory will be executed.

If the RUN command is given with just a *filespec\$* (i.e. no additional parameters are specified), no variables will be retained, and any open files will be closed.

If the Interpreter BASIC "RUN" command is given with the line number parameter, the program specified will be loaded, and execution will begin at the line specified. Realize that the line number specified must be an existing line number, or an Undefined line number error will be generated. Also, it must be represented as a numeric constant. If a line number is not specified, execution will begin with the first line number of the program.

The "RIV" parameters are optional, and are used primarily when BASIC programs are to be "chained" together. One of two different parameters are available. Either "R" or "V" may be specified, but not both. If used, they must be represented as a letter ("R" or "V"), and cannot appear within quote marks, or be represented by a string expression.

If the RUN command is given with the "R" parameter, all variables will be lost, but any files which were opened will remain open, and will utilize the same buffer number. Realize that if the "R" parameter is used, any open files must be re-fielded.

If the RUN command is given with the "V" parameter, any established variables will be maintained, and all open files will remain open. There are several points to be considered when using the "V" parameter. In addition to all files remaining open, the fielding of the buffer associated with the open file will remain intact. Hence, re-fielding is not required. Any DEFinition statements (such as "DEFINT" and "DEFSTR") must be re-established in the program to be chained. The "CLEAR" command should not be encountered in the program to be chained, as execution of a "CLEAR" statement will close all open files and destroy any established variables.

It should be obvious that if the program to be chained is longer than the calling program, or uses more variables than the calling program, an Out of memory or Out of string space error may occur. To utilize this feature to its fullest capabilities, forethought must go into the determination of variable names to be carried over from one program to another.

The "RIV" and "line number" parameters may be specified individually, or they may appear together in the RUN command. If both parameters are

specified, the "RIV" parameter must physically come before the *line number* parameter.

Suppose you have a BASIC program named MYPROG/BAS, and this program has been saved onto a disk which is currently in drive :1. One of the following commands may be given to load and execute the above program.

```
RUN"MYPROG/BAS:1"  
A$="MYPROG/BAS:1":RUN A$
```

After either of the above commands are executed, any program currently in memory will be overwritten, and the program MYPROG/BAS will be loaded and executed. Any open files will be closed, and any established variables will be destroyed.

Suppose you wish to load and execute the program MYPROG/BAS as described in the above example, except that you wish execution to begin at line 3000 in the program. The following command will cause the program to be loaded, and execution will begin at line 3000.

```
RUN"MYPROG/BAS:1",3000
```

This example will illustrate how to use the "V" parameter of the RUN command to maintain variables between chained programs. Listed below will be two programs that reference each other (PROG1/BAS and PROG2/BAS). The sequence will be started by issuing the command RUN"PROG1/BAS". Both programs must have been saved on disk prior to trying to execute either.

```
5 'PROG1/BAS  
10 CLEAR 2000  
20 DEFINT A-Z:DEFSTR S  
30 IF A=0 THEN S="PROG1/BAS"  
40 CLS  
50 A=A+5  
60 PRINT"This is ";S,"A=";A  
70 IF A>100 THEN END  
80 S="PROG2/BAS"  
90 INPUT"Press <ENTER> to run PROG2/BAS";S1  
100 RUN"PROG2/BAS",V,20
```

```
5 'PROG2/BAS
10 CLEAR 2000
20 DEFINT A-Z:DEFSTR S
30 CLS
40 A=A+3
50 PRINT"This is ";S,"A=";A
60 S="PROG1/BAS"
70 INPUT"Press <ENTER> to run PROG1/BAS";S1
80 RUN"PROG1/BAS",V,20
```

Notice that in each of the "RUN" commands, line number 20 was specified. This accomplishes two things. It causes execution to start at *line 20* of each program, which will cause the "CLEAR" command in both programs to be bypassed. Also, line 20 must be executed, as all DEF-type statements must be re-established when programs are chained using the "V" parameter. Although this is a very simplistic example, it should illustrate some of the steps needed to perform program chaining while retaining variable assignments. If both the "V" parameter and a line number are used, the "V" parameter must come before the line number.

### Compiler BASIC RUN

RUN loads and runs a machine language program (executable CMD file) from disk. It can be any executable program including another compiled program. The "RUN" statement also supports additional variations beyond RUN"filespec". For example, the statement,

#### RUN

will restart your program without causing it to load. All open files will be first closed. All variables will be cleared. The statement,

#### RUN exp

will restart your program at the line identified by *exp*. All variables will be cleared. All open files will be closed. This is similar to interpretive BASIC's, "RUN nnnn". Of course, if the syntax used is, "RUN exp\$", then *exp\$* is assumed to be a file specification. You cannot "RUN" at a labeled statement as there can be no differentiation between a labeled statement string and a filespec string.



# SCALE

# SCALE

This compiler BASIC statement is used to establish a scaling factor for the "DRAW" statement.

## Compiler BASIC

**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. Here's a modification of the "DRAW" illustrating a rotating figure drawn at four different scales.

```
10 DEFINT F:CLS:DIM FIGURE1(110)
20 Y=0:FOR X=0 TO 250 STEP 10
40 FIGURE1(Y)=X*6:'Line length = 6*X/256 units
50 FIGURE1(Y+1)=X+256:'Rot = X, entry code = 1
55 Y=Y+2
60 NEXT:'        Continue until figure completed
70 FIGURE1(Y+1)=0:'0 entry to terminate list
75 ' Draw it!
   FOR I = 0 TO 255 STEP 16:ROT=I:'Rotate figure
   FOR J = 4 TO 1 STEP -1
   SCALE = 64*J: 'Scale figure
80 DRAW SET @64,23 USING FIGURE1(0)
   DRAW RESET @64,23 USING FIGURE1(0)
   NEXT J:NEXT I
   PRINT@80*11+30,"That's all folks":A$=WINKEY$
```

# SET

# SET

This statement is used to turn on a pixel.

## Compiler BASIC and DOS 5 Interpreter BASIC

### SET(x,y)

### Statement

- |          |   |
|----------|---|
| <b>x</b> | Is a numeric expression which evaluates to the range <0 - 127> for 64-column screens and <0 - 159> for 80-column screens. |
| <b>y</b> | 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. A function, "POINT(x,y)", is also related to the pixel graphics commands. Note that screens that display 16 rows of 64 characters will display 72 rows by 160 columns of graphic's pixels; screens that display 24 rows of 80 characters will display 72 rows by 160 columns of graphic's pixels.

The following illustrates a brief example of these graphics commands:

```
5   Y=23:RANDOM:CLS
10  FOR X=0 TO 127: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:COMPL(X,23):NEXT
80  FOR X=0 TO 127: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.

## SETEOF

## SETEOF

The SET EOF command may be used to “shrink” the amount of space taken up by a file by truncating it, and thus free-up additional disk space. The syntax for the SET EOF command is:

<b>SET EOF</b> <b>bufnum</b>	Statement 15
<b>%SETEOF(bufnum)</b>	Statement C
<b>bufnum</b>	Represents the buffer number used to open the file in question, and can be expressed as an integer constant or an integer expression.

The SET EOF command is used primarily in conjunction with random files. In some applications, a random file may contain unwanted records at the end of the file. The “SET EOF” command will furnish you with a way to eliminate these unwanted records. The function it performs is to reset the end of file marker for the file in question to a value less than the current end of file marker. This will cause all records whose record numbers are greater than the new end of file marker value to be ignored, and thus make these records inaccessible. Also, the space taken up on the disk by these “eliminated” records will be added to the free space available, and thus may be reused.

To use the “SET EOF” command, you must open the file in question as a random file (i.e. type “R”). It is highly recommended that the record length used to open the file be the same as the record length used for normal access to the file.

After the file has been opened, perform a “GET” of the record you wish to be the last record in the file. You may then use the “SET EOF” command to reset the end of file marker to the current record number, and thus eliminate all unwanted records (by doing a “GET”, the current record number will be changed to the value of the record which was retrieved).

Suppose you have a random file named XTRA/DAT which currently contains 100 records, and you wish to eliminate the last 50 records of the

file (records 51-100). Assume also that the file has been opened in the random mode, using buffer number 3. The following commands may be used to accomplish this "file shrinkage".

### GET3,50:SET EOF3

Be extremely careful when using the SET EOF command. Once records have been eliminated from a file using this command, they might not be recoverable! It is beyond the scope of this manual to discuss techniques used to recover lost information in a file. The best prevention for such an occurrence is caution!

Compiler BASIC does not directly support "SET EOF"; however, the "COMMAND" facility may be used to create a "%SETEOF" command; this is demonstrated following the detailed example.

Here's a more detailed example program illustrating an easy user command designed to set the "EOF" of a random access file. The random file is positioned to the last record desired by means of a "GET" on that record. Then the "SETEOF" [of course coded as **%SETEOF(bufnum)** since Compiler BASIC requires the "%" character to indicate a user command] will set the "EOF" pointers to the current record pointers.

```
ALLOCATE 1:      'Allocate one file buffer space
OPEN "R",1,"testfile/dat",32
FIELD 1,32 AS ARG$
FOR I = 1 TO 20
  LSET ARG$="This is test "+STR$(I)
  PUT 1,I: NEXT
CLOSE 1
SYSTEM"List testfile/dat (hex)"
OPEN "r",1,"TESTFILE/DAT",32
FIELD 1,32 AS ARG$
GET 1,10: PRINT ARG$
%SETEOF(1):      'Set the end of file
CLOSE 1
END"list testfile/dat (hex)"
*INCLUDE SETEOF/BAS
```

The %SETEOF(bufnum) command code is not an integral part of Compiler BASIC; it can be written to a file named "SETEOF/BAS". This file must be included with your program in order to be able to use the command. Here's the code:

```
COMMAND SETEOF (BUFNUM%)
IF NOT EOF (BUFNUM%)
Z80-MODE
LD HL, (& (BUFNUM%)) :CALL @CALADR
LD A, (IX+16+5) :LD (IX+16+8), A
LD A, (IX+16+10) :LD (IX+16+12), A
LD A, (IX+16+11) :LD (IX+16+13), A
HIGH-MODE
ENDIF
RETURN
ENDCOM
```

If you care to examine that file, you will see that it consists, in part, of a Z80-MODE routine. "@CALADR" is a routine from the Compiler BASIC SUPPORT/DAT library which calculates the address of a bufnum's file buffer allocation and returns the value in register IX. To be fancy about it, the CF would be set on return from @CALADR if the bufnum referenced a file which was not already open. Of course, under that case, nothing damaging would result as the accessing of the FCB region would be of unused memory addresses. Make note that if the bufnum exceeds the value established by ALLOCATE (the maximum number of open files), @CALADR would report a runtime error 104.

# SGN

# SGN

This function obtains the sign of its argument.

Compiler BASIC and Interpreter BASIC	
<b>SGN(exp)</b>	Function
<b>exp</b>	Is a numeric expression.

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

$\text{SGN}(\text{exp}) = -1$  if  $\text{exp} < 0$

$\text{SGN}(\text{exp}) = 0$  if  $\text{exp} = 0$

$\text{SGN}(\text{exp}) = 1$  if  $\text{exp} > 0$

# SIN

# SIN

This function obtains the trigonometric sine of its argument.

Compiler BASIC and Interpreter BASIC	
<b>SIN(exp)</b>	Function
<b>exp</b>	Is a numeric expression in radian measure.

**SIN** takes the sine, in radians, of an expression. Interpreter BASIC returns a single precision value; Compiler BASIC 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.

# **SORT,    SCLEAR,    KEY,    TAG**

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

## **Compiler BASIC**

<b>SORT [(flag),] num</b>	Statement
<b>SCLEAR</b>	Statement
<b>KEY array(exp)</b>	Statement
<b>TAG array(exp)</b>	Statement

**array(exp)**    Is an array element which designates the key array for sorting purposes and the tag array for sorting purposes.

**num**            Is an integer numeric operand in the range (1 to 32767) which designates the number of elements to sort.

**flag**           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 compiler "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.

# SOUND

# SOUND

This statement generates a sound using a DOS service call, when available. Its syntax is:

DOS 6 Compiler BASIC and DOS 6 BASIC	
<b>SOUND tone,duration</b>	Statement 6
<b>%SOUND(tone,duration)</b>	Statement C
<b>tone</b>	Is a number specifying the tone in the range <0-7>
<b>duration</b>	Is the duration of the tone entered as an integer in the range <0-31>.

The **SOUND** command generates a tone from the computer's speaker (tone board). It relies on the **SOUND** support available from DOS. *Tone* varies in frequency with "0" representing the lowest and "7" representing the highest. The *duration* can range from the shortest ("0") to the longest ("31"). **SOUND** is best used as an alert tone for calling attention to an error encountered or a required entry while running a program.

Compiler BASIC does not internally support any "SOUND" statement. On the other hand, a little "COMMAND" addition in Z80-MODE should allow you to adapt that Russian Roulette program very easily. Here's a little code which adds a **SOUND** statement and demonstrates its use:

```
DEFINT A-Z
FOR D=0 TO 4
FOR T = 0 TO 7
  %SOUND (T,D)
NEXT T,D
STOP
*INCLUDE SOUND/BAS
```

You would need to include the file named "SOUND/BAS" which contains the code necessary to generate the sounds. It uses the DOS @SOUND supervisor call. A statement such as the fourth line will beep

the requested sound. Although this syntax "SOUND(tone,duration)" is a little different from "SOUND tone,duration" in DOS 6 Interpreter BASIC, it's a straightforward conversion.

Here's the SOUND/BAS code which is a Z80-MODE routine. This is another example of the ease in which an assembly language routine can be integrated into your BASIC programs without USR, CALL, or packed strings.

```
COMMAND SOUND (TONE,DUR)
Z80-MODE
LD A, (& (TONE)) :AND 7:LD B,A
LD A,7:SUB B:LD B,A
LD A, (& (DUR)) :AND 31
RLCA:RLCA:RLCA:OR B:LD B,A
LD A,104:RST 40:RET
HIGH-MODE
ENDCOM
```

## SPACE\$

## SPACE\$

This DOS 6 Interpreter BASIC function returns a string of "SPACE" characters. It's syntax is:

DOS 6 Interpreter BASIC	
SPACE\$(number)	Function
number	The quantity of spaces desired in the range <0-255>.

The SPACE\$ function returns a string filled with as many spaces as the operand, *number*. It is primarily useful in "PRINT" statements, or in initializing string variables to SPACE characters, CHR\$(32).

```
10 A$ = SPACE$(30)
20 PRINT A$;"This prints at column 30"
```

The above program assigns a string of 30 SPACE characters to the variable "A\$", then prints the message starting at column 30 (relative to column 0).

## SPC

## SPC

This DOS 6 Interpreter BASIC function prints a string of SPACE characters. Its syntax is:

DOS 6 Interpreter BASIC	
<b>SPC(number)</b>	Function
<b>number</b>	The quantity of spaces desired in the range <0-255>.

SPC is similar to "SPACE\$"; however, it can be used only with the "PRINT", "LPRINT," or "PRINT#" statements, and it uses no string space.

```
10 PRINT SPC(30)"This prints at column 30"  
20 PRINT SPC(30);"This also prints at column 30"  
30 PRINT SPC(30),"But this does not!"
```

Omitting a terminating character after the closing parenthesis is the same as terminating the function with a semi-colon. Using a comma terminator positions the next printed character at the tab-stop following the spacing.

## SQR

## SQR

This function obtains the square root of its argument.

Compiler BASIC and Interpreter BASIC	
<b>SQR(exp)</b>	Function
<b>exp</b>	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,  $\text{SQR}(4) = 2$ , since  $2 * 2 = 4$ , and  $\text{SQR}(81) = 9$ , since  $9 * 9 = 81$ . Usually the result is not a neat integer, as with  $\text{SQR}(7)$  (= approximately. 2.64575).

Interpreter BASIC returns a single precision result; in compiler BASIC, a double precision expression will cause a double precision square root to be returned, accurate to at least 16 decimal digits.

"**SQR(A)**" will return an illegal function call error if "**A**" is negative.

```
1 ON BREAK GOTO 100: ON ERROR GOTO 200
2 PRINT"<BREAK> transfers to square root"
3 PRINT"where <BREAK> terminates program."
10 INPUT "A";A
20 INPUT "B";B
30 C=A^B
70 PRINT"A^B=";C
80 PRINT:PRINT
90 GOTO 1
100 ON BREAK GOTO 0
110 INPUT"Enter Z (<BREAK> terminates)";Z
120 Y=SQR(Z)
130 PRINT "SQUARE ROOT OF Z =" ;Y
140 PRINT:PRINT
150 GOTO 1
200 PRINT "Error ";ERR;" in line ";ERL:RESUME 2
```

# STOP

# STOP

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

Compiler BASIC and Interpreter BASIC	
STOP	Statement

**STOP** in Interpreter BASIC will cease the execution of the running program and print the message, **BREAK IN line**, where *line* is the line number where the “STOP” occurred. While the program is stopped, you may examine the values of variables or change their contents; you cannot edit any lines or the program’s variables will be cleared. The “CONT” statement may be used to resume execution.

**STOP** in Compiler BASIC 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.



**STR\$****STR\$**

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

Compiler BASIC and Interpreter BASIC	
<b>STR\$(exp)</b>	Function
<b>exp</b>	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.

# STRING\$

# STRING\$

This function generates a repeated character string.

Compiler BASIC and Interpreter BASIC	
<b>STRING\$(exp1,exp2)</b>	Function
<b>STRING\$(exp1,"char")</b>	Function
<b>exp1</b>	Is equal to the desired string length.
<b>exp2</b>	Is equal to a character code in the range <0-255>.
<b>"char"</b>	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

# SWAP

SWAP is a BASIC statement used to exchange the contents of two similarly typed variables.

Compiler BASIC and DOS 6 Interpreter BASIC	
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".

# SYSTEM

# SYSTEM

This statement is used to invoke a DOS command from within a program and then return to your program when the DOS command completes. It is used under DOS 5 Interpreter BASIC to load from cassette tape, or transfer control to, machine language programs.

## Compiler BASIC and Interpreter BASIC

<b>SYSTEM "command"lvar\$</b>	Statement	C
<b>SYSTEM ["command"lvar\$]</b>	Statement	16
<b>SYSTEM</b>	Statement	15
<b>CMD "command"lvar\$</b>	Statement	15

**command** Is the string to pass to DOS for execution;  
if omitted under DOS 6 Interpreter BASIC, the  
program will stop and control is returned to DOS.

**var\$** Contains the command string.

Under DOS 6 Interpreter BASIC and Compiler BASIC, **SYSTEM** will pass the command string identified as the parameter to the DOS command interpreter. Upon completion of the command, control will be returned to the running program. Under DOS 5 Interpreter BASIC, the equivalent operation is invoked with the **CMD"command"** statement.

Under Interpreter BASIC, omitting the parameter will exit the program and return back to DOS. If you use the *command* parameter, the command string is restricted to reference DOS library commands only. Actually, the restriction is that the command invoked must not utilize any memory outside of the DOS library region (2400H-2FFFFH). If you have specific knowledge that a non-library command supports this restriction, you may invoke it using the DOS's "RUN" command; i.e. **SYSTEM"RUN MYPROG"**. The Interpreter BASIC BASIC/OV1 file must be available to enable use of "SYSTEM".

Compiler BASIC has no restriction on invoking any command, short of commands which can alter the high-memory pointer.

The following DOS 6 Interpreter BASIC example illustrates the use of **SYSTEM"RUN command..."** to sort arrays using a **BSORT** utility.

```
10 OPTION BASE 0: DIM A$(6), F$(6)
20 FOR I=1 TO 6: READ A$(I): NEXT
30 FOR I=1 TO 6: READ F$(I): NEXT
40 SYSTEM"RUN BSORT 6, A$(1), +F$"
50 FOR I=1 TO 6
60 PRINT A$(I), F$(I)
70 NEXT
80 END
90 DATA SMITH, JONES, JONES, WILLIAMS, JOHNSON, JONES
100 DATA SAMMY, BILLY, BETTY, RICHARD, CHARLES, BOBBY
```

Under DOS 5 Interpreter BASIC, you use **SYSTEM** to load and/or execute a machine language program from cassette tape. After typing **SYSTEM** <ENTER>, you will receive a question mark prompt. Enter the name of the tape file you wish to load followed by <ENTER>. After the file is successfully loaded, another question mark will be displayed. Execution can be started at the default address stored with the program by typing a slash, "/" followed by <ENTER>.

You may begin execution at any address by typing a slash followed by the memory address to begin execution.

Note that to access cassette tapes, you must disable interrupts first (see **CMD"T"** and **CMD"R"**). Note also that most system tapes will overlay the DOS and cannot be utilized while running **DISK BASIC** (see **"CMDFILE"** in your DOS manual).

## SZONE

## SZONE

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

Compiler BASIC	
<b>SZONE(pos 1,...,pos n)</b>	Statement
<b>SZONE(*)</b>	
<b>pos</b>	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. Any screen print zone established remains until all are cleared; thus, the “**SZONE**” statement may be repeatedly invoked to establish more than one zone, or more than one screen tab position may be passed as operands of the statement. **SZONE(\*)** clears all print stops. See the program below for an example “**SZONE**” usage, as well as “**PZONE**” for similar use with line printer “**LPRINT**” zones.

```
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.

# TAN

# TAN

This function obtains the trigonometric tangent of its argument.

Compiler BASIC and Interpreter BASIC	
<b>TAN(exp)</b>	Function
<b>exp</b>	Is a numeric expression in radian measure.

TAN returns the radian degree tangent of an expression, mathematically equivalent to "SIN(exp)/COS(exp)". Interpreter BASIC returns a single precision value. Compiler BASIC will return a double precision value if the given expression is double precision.

## TIME\$

## TIME\$

The **TIME\$** function will retrieve the current system time (and date for DOS 5 Interpreter BASIC) as a string. The syntax for the “**TIME\$**” command is:

Compiler BASIC and Interpreter BASIC		
<b>TIME\$</b>	There is no operand	Function

Under DOS 6 Interpreter BASIC and Compiler BASIC, the system time is returned as an eight-character string of the form, “HH:MM:SS”. Under DOS 5 interpreter BASIC, both the system date and time is returned as a 17-character string of the form: “MM/DD/YY HH:MM:SS”. *MM*, *DD*, and *YY* represent the month, day of the month, and year respectively, as kept by the operating system. The *HH*, *MM*, and *SS* represent the hours (00-23), minutes (00-59) and seconds (00-59) respectively, as retrieved from the real time clock when the “**TIME\$**” command was actually executed. The slashes (“/”) and colons (“:”) will always be present in the string, and a space will always separate the date information from the time information.

The value returned from the “**TIME\$**” command can be used in a manner similar to the value returned from the “**MEM**” function. It may be used directly (as in the statement **PRINT TIME\$**), or may be assigned to a string variable (as in **A\$=TIME\$**). The value returned by the “**TIME\$**” command will always be an eight-character or seventeen-character string.

```
CLS
PRINT TIME$
FOR I = 1 TO 10000 : NEXT I
PRINT TIME$
```



## TROFF, TRON

## TROFF, TRON

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

<p><b>Compiler BASIC and Interpreter BASIC</b></p>
--

<p><b>TROFF</b> <b>TRON</b></p>
-------------------------------------

During program development, it may be useful to trace the execution flow of your program during debugging. TRON and TROFF allow you to selectively control where in your program trace information will be generated. After a "TRON", the line number of each statement executed will be displayed surrounded by braces. This trace information will cease upon execution of the "TROFF" statement.

Under Compiler BASIC, "TRON" acts similarly to interpretive BASIC TRON. However, 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. See the "NX" and "YX" compiler directives for additional information.

# TYPE

# TYPE

This Compiler BASIC function obtains the type code of its argument.

Compiler BASIC	
<b>TYPE(exp)</b>	Function
<b>exp</b>	Is a numeric or string expression.

TYPE returns the variable type code of the expression. These type codes are as follows (Interpreter BASIC *type* codes are shown for comparison):

Variable Type	Compiler Code	Interpreter
integer	1	2
single precision	2	4
double precision	4	8
string	3	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,  $\text{TYPE}(\text{A}\$(0)) = 128 + 16 * 1 + 3 = 147$ . Note that the array index ("0") is arbitrary; it need only be within the dimensioned range.

# UP

# UP

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

Compiler BASIC	
UP	Statement

**UP** scrolls the entire screen up by one line, clearing the bottom line. This is equivalent to the “standard” screen scroll.

# USING

# USING

This compiler BASIC function is used to define formatted PRINT output.

## Compiler BASIC and Interpreter BASIC

**USING format\$;explist**

Function

**format\$**    Is the format control string.

**explist**    Is the expression list.

The compiler BASIC "USING" string function works equivalently to Interpreter BASIC's, "PRINT USING"; however, compiled "USING" provides the ability to store and manipulate "USING" formatted data with string handling instructions as its value may be assigned to a string for subsequent manipulation. That makes this implementation much more versatile than the "PRINT USING" scheme. Consider this brief example:

```
A$=USING "###.## ";1.666,345.555,17.893
PRINT LEFT$(A$,12):PRINT A$:W$=WINKEY$
```

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. The length of the format string is limited to 63 characters under DOS 5 and 79 for DOS 6.

"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. For additional details, see "PRINT USING" on page 213

## USR

## USR

The USR statement will allow a BASIC program to branch to a user written machine language subroutine. The syntax for the "USR" command is:

Interpreter BASIC	
<b>variable=USRn(intval)</b>	Function 15
<b>variable=USRn(exp)</b>	Function 16
<b>variable</b>	Must be a numeric variable, and in most cases should be of integer type. If a value is to be returned from the machine language subroutine, it may be contained in this variable when the machine language routine returns to BASIC.
<b>n</b>	Is the user routine number (0-9) used to identify the routine in question (user routines are defined with the DEFUSR command). The routine <i>number</i> must be represented as a numeric constant.
<b>intval</b>	Is a value which will be passed to the user machine language subroutine. It may be represented as a numeric expression or a numeric constant, and must be expressed as an integer value.
<b>exp</b>	Is a value which will be passed to the user machine language subroutine. It may be represented as any string or numeric expression.

The USR command will allow you to jump to a machine language subroutine from within your BASIC program. The machine language subroutine will generally be resident in high memory, and the memory used by the module must be protected, either using the DOS MEMORY command, or by specifying the "M" parameter when entering BASIC.

Prior to issuing a "USR" call, the starting address of the specific machine language subroutine must have been defined using the "DEFUSR" command.

Once the "USR" call is performed, execution of your BASIC program will be halted, and a jump will be done to the address specified in the corresponding "DEFUSR" statement. Your machine language subroutine will then take over, until a return to BASIC is performed in the machine language module. Once this return to BASIC has been encountered, your BASIC program will regain control.

### Initiating a USR call

Suppose you have loaded and protected a machine language module. In addition, you have defined this machine language module with the following command:

```
DEFUSR5=&HF400
```

To perform a jump to this machine language module, the following command may be given:

```
XX%=USR5(1024)
```

Upon executing the above command, execution of the BASIC program will be halted, and the machine language instruction at address X'F400' will be executed. The value 1024 will be passed to the machine language routine. The machine language routine will continue to be executed, until a return to BASIC is encountered. If any value is to be returned from the subroutine, it will be contained in the integer variable XX% when BASIC regains control.

### Passing values to and from machine language subroutines

In the above example, the value 1024 was passed to the machine language subroutine. In order to utilize this value in the subroutine, the first code block of the machine language routine should be one of the following:

```
CALL 0A7FH
```

```
DOS 5
```

---

## BASIC Statements and Functions

---

```
LD      HL, RETADR          DOS 6
PUSH    HL
LD      HL, (2603H)
JP      (HL)
RETADR  ...
```

Executing the above code block as the first statement in the machine language subroutine will cause the value 1024 to be placed in the HL register, with "H" containing the MSB, and "L" containing the LSB of the value.

When BASIC passes control to the USR, the Z80 registers contain values you use to recover the argument value passed. Since DOS 5 BASIC allows only an integer argument, and DOS 6 BASIC allows any type of argument, these register values differ; they are set up as follows:

Register	Contents	
A	Type of value passed: 2, 3, 4, or 8	16
HL	Pointer to numeric argument value	
DE	Pointer to string DCB if type=3	16

For integers, HL points to the Least Significant Byte (LSB); for single precision, it's the LSB of the mantissa; for double precision, it's the third most significant byte.

To return a value from a machine language subroutine to BASIC, you should use one of the following code blocks as the last statement in your subroutine:

```
JP      0A9AH              DOS 5

LD      HL, (2605H)        DOS 6
PUSH    HL
LD      HL, VALUE
RET
```

After your machine language module executes the above code block, control will return to BASIC (the statement following the USR call), and the variable used in the "USR" call will be assigned the value that was in the HL register pair prior to the JP or RET command. If no value is to be returned from your machine language module, you may use a RET command to return to BASIC.

## User Interface to SVCs

A user interface to the DOS SVC functions is provided under DOS version 6 Interpreter BASIC via the "USR" statement in BASIC. To use this interface, establish an integer array with the first six elements containing the following information that may be needed by the SVC:

Element 0:	SVC number (Always needed!)
Element 1:	Value for register pair HL
Element 2:	Value for register pair DE
Element 3:	Value for register pair BC
Element 4:	Value for register pair IY
Element 5:	Value for register pair IX

The interface is accomplished by using a normally out of range "USR" argument, USR11. To execute the SVC, use the syntax:

**USR11(VARPTR(ARRAY(0)))**

For clarity, "ARRAY" has been used for the name of the array in the example, but the actual name of the array used to pass the parameters to USR11 must be only one or two characters long. If the name of the array is longer than two characters, USR11 can't check the array type bit because of some technical details of the way "VARPTR" works. No "DEF USR" statement is required. After the SVC executes, the register pairs will be unloaded back into the array. The AF register pair will be placed in array position "0". If the array is not an integer type, or if the SVC number is either zero or greater than 127, an illegal function call error will occur.

The return condition of the SVC can be tested by checking the bits in ARRAY(0). Doing an "AND 64" will produce a non-zero value if the Z flag is set. Doing an "AND 1" will produce a non-zero value if the Carry Flag is set. For further explanation of DOS SVC usage and returned values, refer to the *DOS Programmer's Reference Guide*. Following is a short example of using the SVC interface.

```
100 DEFINT J,K: DIM J(5) ' Important - integer
    array only for SVC interface, must be a 1 or 2
    character name
200 CLS: PRINT TAB(25) "SVC Demonstration Menu"
210 PRINT: PRINT TAB(25) "1) Set Scroll Protect"
220 PRINT TAB(25) "2) Toggle Caps Lock"
```



## BASIC Statements and Functions

---

```
230 PRINT TAB(25)"3) Show DOS Version"
240 PRINT TAB(25)"4) Check Drive Ready"
250 PRINT:PRINT TAB(25);:INPUT "Make a selection
";A$
260 IF A$ [ "1" OR A$ ] "4" THEN 200
270 A=VAL(A$):ON A GOSUB 1000,2000,3000,4000
280 GOTO 200
1000 CLS:PRINT"Set number of scroll protect
lines 0-7"
1010 PRINT:PRINT"0 will cancel scroll protect
";:INPUT A
1020 IF A [ 0 OR A ] 7 THEN 1000
1030 J(3)=&H700 + A 'Register B=7, Register
C=line count
1040 J(0)=15:X=USR11(VARPTR(J(0))) 'Execute
@VDCTL SVC
1050 IF A=0 THEN RETURN
1060 FOR K=1 TO 100:PRINT K;"Testing scroll
protect to 100":NEXT:RETURN
2000 CLS:J(0)= 101:X=USR11(VARPTR(J(0))) '
@FLAGS SVC to get keyboard flag
2010 K=PEEK(J(4)+10)' Get KFLAG value
2020 POKE (J(4)+10), K XOR 32 'Toggle Caps lock
bit
2030 INPUT"Type some characters to check Caps
lock, press ENTER to end ";A$
2040 RETURN
3000 CLS:J(0)=101:X=USR11(VARPTR(J(0)))' Get
flag table base, @FLAG SVC
3010 K=PEEK(J(4)+27)' Version number, base+27
3020 K=K-(&H60)' Earliest version was 6.0, K
will be 0 to 3
3030 PRINT "This is DOS version 6 .";K
3040 PRINT:INPUT"Press ENTER to return";
A$:RETURN
4000 CLS:INPUT"Enter drive # to check 0 to 7 ";K
4010 IF K [ 0 OR K ] 7 THEN RETURN
4020 J(3)=K' Set drive number for SVC
4030 J(0)=33:X=USR11(VARPTR(J(0)))' @CKDRV SVC
4040 PRINT:PRINT"Drive";K;"has ";
4050 IF (J(0) AND 64)=0 THEN PRINT"no disk
mounted.":GOTO 4080
4060 PRINT"a disk mounted."
```

```
4070 IF (J(0) AND 1)=1 THEN PRINT"The drive is  
write protected."  
4080 PRINT:INPUT"Press ENTER to return  
";A$:RETURN
```

This example program shows several methods of accessing information not normally available from BASIC. To make effective use of the SVC interface, you will need to have the *Programmer's Reference Guide*.

The @CKDRV routine (lines 4000-4080) shows how the return status of an SVC can be checked. The Z flag can be tested by doing an "AND 64" against the array(0) position. If the result is zero, the Z flag was NOT set. The CF (carry flag) can be tested in the same manner by doing an "AND 1". Again, a zero result means the flag was not set.

The Caps lock and DOS Version subroutines show how information can be referenced in the system flag area with the @FLAGS SVC. There are several other flags and system storage areas that can also be referenced off of the flag table base.

Certain SVCs, such as @CMNDI, @CMNDR, or setting the high memory pointer must not be done. Generally, anything that can legally be done from BASIC with @CMNDR can be done with the SYSTEM "command" statement, and should not be attempted through the SVC interface.

### **Brief overview of BASIC SVC access of video screen**

The following program provides a brief exposure to accessing the Model 4 video screen contents using the @VDCTL service call of DOS 6.3 and the USR11 interface provided in BASIC..

```
100 DEFINT J,K:DIM J(5), S$(24): REM S$ holds 24  
screen lines  
110 RX$="01234567890123456789012345678901234567890123456789"  
901234567890123456789012345678901234567890123456789"  
200 IR=VARPTR(RX$): REM get loc'n of buffer DCB  
210 IF IR < 0 THEN IR = IR + 65536!: REM Adjust  
DCB location  
220 IL = PEEK(IR+1):IH=PEEK(IR+2): REM Get  
address of string constant  
250 CLS:FOR K = 0 TO 22: PRINT "This is a test  
of line ";K:NEXT
```

```
300 FOR K = 0 TO 23: REM Issue @VDCTL, function
9, 24 times
310 J(0) = 15: REM @VDCTL service call
320 J(1) = K * 256: REM Set row number into
register H
330 POKE VARPTR(J(2)),IL:POKE VARPTR(J(2))+1,IH:
REM Our 'buffer' into DE
340 J(3) = 9 * 256 + 1: REM function 9, video
line to buffer
350 X = USR11(VARPTR(J(0))): REM Invoke the SVC
360 S$(K) = LEFT$(RX$,80): REM Must use LEFT$ to
copy characters
370 NEXT
400 CLS:FOR K = 0 TO 23: PRINT S$(K);:NEXT: REM
Show what we read from screen
500 FOR K = 0 TO 79
510 J(0) = 15:J(1) = K:J(3) = &H100
540 X=USR11(VARPTR(J(0)))
550 C=(J(-) AND &HFF00)/256
560 PRINT CHR$(C);
570 NEXT
```

Line 100 declares the integer array, J(5), used to pass Z80 register contents to/from the system SVC. The string array, S\$(24), is declared and will be used to store the 24 lines read from the screen.

Line 110 declares a string constant and initializes it so that it is 80 characters long. In order to use the line at a time video transfer function of @VDCTL, the buffer must be below X'F400', the area of memory swapped out to perform video and keyboard access. Using a string constant guarantees that this 80-byte region will be sufficiently low in memory to avoid any address complications.

Lines 200 and 210 obtain the address of the string constant's DCB and adjust it to a positive value, as required.

Line 220 obtains the actual address of the string constant as stored in memory; the low and high order address bytes are preserved so that they may be transferred to the integer array of data passed to the SVC. Too bad BASIC has no WPEEK function!

Line 250 clears the video screen and displays 23 lines of data so that we know exactly what is on the screen. This is done here for illustration purposes only.

Lines 300-370 form the loop to issue the @VDCTL SVC 24 times; each time a single screen line will be transferred.

Line 310 initializes the SVC number desired into the integer array.

Line 320 passes the desired row number into the integer array. Since the row number is stored in register H, we must shift the small integer 0-23 left by 8 bit positions to align the value into register H. This is done by multiplying the loop index value by 256.

Line 330 pokes the address of the string constant (our line buffer) into the integer array for the "DE" value.

Line 340 establishes the VDCTL request as function 9 (transfer a 80-byte video line) and specifies the direction to be screen to buffer.

Line 350 invokes the @VDCTL SVC by using the USR11 function.

Line 360 copies the result of the video line transfer into our string array which holds the 24 lines, each line a separate array element. Note that the LEFT\$ function is being used to force a copy of the characters. If the program used a simple assignment statement, such as S\$(K)=RX\$, only the string address pointer in the S\$ array would be altered since BASIC assumes that string constants are not going to be changed; new strings would not be generated and each array element would actually contain the same result - the last line transferred.

Line 400 simply displays the result of the program's capture of the screen lines.

Lines 500-570 read the top line of the video screen by the single character "peek" function of @VDCTL and displays the result of each character read.

# VAL

# VAL

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

Compiler BASIC and Interpreter BASIC	
<b>VAL(exp\$)</b>	Function
<b>exp\$</b>	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.

# VARPTR

# VARPTR

This function obtains the absolute memory address of its argument.

## Compiler BASIC and Interpreter BASIC

**VARPTR(var)**

Function

**VARPTR(#bufnum)**

Function I

**var**            Is any numeric or string variable or array element.

**bufnum**        Is an Interpreter BASIC file buffer number.

**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 the case of numeric variables, or the address of the string data control block in the case of string variables. These values may be useful to pass variable data to a "USR" routine. Data control blocks for strings and arrays differ between Interpreter BASIC and Compiler BASIC; the storage structure of these control blocks is listed in the *Technical Section*.

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

**PEEK(VARPTR(T\$)) + 256 \* PEEK(VARPTR(T\$)+1)**

For example, supposing that the "LEN" function didn't exist in Compiler BASIC. Then:

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

This 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 Compiler BASIC for user defined functions and is explained in the section on FUNCTION ... ENDFUNC.

A variation on the use of Compiler BASIC's "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:

**arrayname()**

Arrayname() returns the address of the array's DCB. For example: TRIALS(), A(), .... See the "*Technical Section*" for details on Data Control Block formats.

# WAIT

# WAIT

This DOS 6 Interpreter BASIC statement will suspend program execution until a specified value is sensed at an input port. Its syntax is:

DOS 6 Interpreter BASIC	
<b>WAIT port,int1[,int2]</b>	Statement
<b>portnum</b>	Is the Z80 CPU port to access; port can be in the range <0-32767>, however the value used is port modulo 256.
<b>int1</b>	A value to AND with the value of the <i>portnum</i> read.
<b>int2</b>	A value to XOR with the value of the <i>portnum</i> read.
Note: execution will continue when the result of the "XOR" and "AND" is non-zero.	

**WAIT** can be used to freeze execution of your program until a particular value is available at a CPU machine port. The *int2* operand may be used to invert selected bits of the port value read, while the *int1* operand may be used to mask selected bits of the port. For instance, to invert all bits of the port, use a value of "-1" for *int2*. Bits 4-7 of the port value may be stripped off (i.e. ignored), by using a value of "15" (1+2+4+8) for *int1*.

## Programmer's Note:

Under Interpreter BASIC, the value of *portnum* can actually range from 0 through 32767; the actual value used for the port determination will be "portnum modulo 256". Thus, a *portnum* of 256 designates port 0; a *portnum* of 257 designates port 1. However, the high-order value will be presented to the high-order address lines of the CPU address bus. This may be useful in addressing internal and external ports of the 64180/Z180 processor which requires a high-order zero value to reference internal ports and a high-order non-zero value to reference external ports.



## WHILE WEND

## WHILE WHEN

These DOS 6 interpreter BASIC statements provide for looping control of a program block. The syntax is:

<b>WHILE</b> <i>exp</i>	Statement 6
statements	
<b>WEND</b>	Statement 6
<b>exp</b>	Is any numeric expression (usually boolean)

**WHILE ... WEND** is a looping construct found in some "structured" languages such as PASCAL. Unlike "FOR ... NEXT" and "REPEAT ... UNTIL", the loop will not be executed if *exp* is initially FALSE (a zero value). If *exp* is non-zero, the block of statements will be executed until the "WEND", then control will branch to the "WHILE" for another evaluation of *exp*. This control flow will continue to repeat until *exp* is FALSE.

"WHILE" ... "WEND" constructs can be nested as "FOR ... NEXT" constructs; each "WEND" will match up with its most recent "WHILE".

Example Program:

```
10 INPUT "Letter (A-Z) to stop for";S$
20 WHILE (T$=CHR$(RND(26)+64)) <> S$
30 PRINT T$,
40 WEND
```

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

## WIDTH

## WIDTH

This DOS 6 interpreter BASIC statement establishes the logical width of the print line. Its syntax is:

DOS 6 Interpreter BASIC	
WIDTH [LPRINT,] size	Statement
<b>LPRINT</b>	Designates the width for the line printer; if omitted, the size refers to the display width.
<b>size</b>	Establishes the width as size.

The use of **WIDTH** is somewhat similar to the *chars* parameter of the DOS FORMS facility. Once you set the width of the printed line to a *size*, a string of characters longer than *size* will wrap to the next line. One significant difference is that if the current print line is not at the first print position, and the string of characters to print is longer than the remaining space until *size*, the printing will start on the next line.

If you have the DOS FORMS filter installed and active to the printer device, and are using width to set the "LPRINT" width, FORMS' *chars* parameter should be set to 255.

```

10 WIDTH 15: 'Set screen width to 15 chars
20 A$=STRING$(40,"*"): 'Gen 40-char string
30 PRINT A$: ' and print it
40 A$=STRING$(10,"#"): 'Gen 10-char string
50 PRINT "Testing";A$: 'Print 7 chars; A$
60 WIDTH 80: 'Reset width to 80 cols
RUN
*****. (wraps to 3 lines)
*****
*****
Testing (only 8 positions left)
##### (forces A$ to new line)

```

## **WINKEY\$**

## **WINKEY\$**

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

Compiler BASIC	
<b>WINKEY\$</b>	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. "WINKEY\$" is essentially equivalent to:

**A\$="":WHILE A\$="":A\$=INKEY\$:WEND**

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
```

# WPEEK

# WPEEK

This compiler BASIC function obtains the two-byte “word” stored at the specified memory address.

Compiler BASIC	
WPEEK(exp16)	Function
<b>exp16</b>	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 or III:

$$\text{V} = \text{WPEEK}(\&\text{H401E})$$

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

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

# WPOKE

# WPOKE

This compiler BASIC statement is used to place a 16-bit word into a memory location.

Compiler BASIC	
<b>WPOKE exp16,exp16</b>	Statement
<b>exp16</b>	Specifies a memory address in the range <-32768 to 32767>.

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

# WRITE

# WRITE

This statement is used to output data to the video screen or sequential access (type "O") disk file.

## DOS 6 Interpreter BASIC

**WRITE[#bufnum,] [item,...]** Statement

**bufnum** Is a file buffer which references an open disk file of type "O" to which the data is to be output.

**item** Is a "string literal" or a numeric / string expression; a list of items may be provided.

"PRINT" is the normal facility used to display data on the video screen. **WRITE** also prints data to the screen, but has some small differences. As in "PRINT", omitting the item list generates an<ENTER>. With a list of items, each data item output, except the last one, would be followed by a comma; an <ENTER> is sent following the last item.

Numeric values which are negative are preceded by a minus sign; however, contrary to "PRINT", positive numbers are not preceded by a blank. Strings are automatically enclosed by quotes.

You may use the "**WRITE#bufnum,list**" form to output the data to a disk file. This facility is useful since it saves you from having to output explicit commas in order to generate *comma-delimited* files.

Note the differences between "PRINT" and "WRITE" in the following:

```
10 DEFINT A-Z
20 READ A,B,C$
30 PRINT A,B,C$
40 WRITE A,B,C$
50 DATA 100,-200,"a string"
RUN
100                -200                a string
100,-200,"a string"
```

## XFIELD

## XFIELD

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

Compiler BASIC	
XFIELD bufnum,var,(exp)var\$,...	Statement
<b>bufnum</b>	Is file control block number, 1-15.
<b>var</b>	Is any non-string scalar variable.
<b>exp</b>	Is the maximum length of the following string variable, <i>var\$</i> .
<b>var\$</b>	Is any scalar 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, an integer takes two bytes to store, a single precision four bytes, a double precision eight bytes, and strings take the specified maximum length (given in the expression in parentheses preceding the string variable name) plus two bytes for the string length. String array elements are not permissible in the "XFIELD" statement.

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 length 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. This particular "XFIELD" statement specifies an integer variable (length 2), a double precision variable (length 8), and a 16-character string (length 16+2). The associated OPEN"X" statement should reference a *reclen* of 28.



## **Technical Information**

### **BASIC Statements**

Note: The codes enclosed in square brackets indicate the BASIC supporting the particular statement. These codes are: I=Interpreter BASIC, I5=DOS 5 Interpreter BASIC, I6=DOS 6 Interpreter BASIC, C=Compiler BASIC, C5=DOS 5 Compiler BASIC, C6=DOS 6 Compiler BASIC. When no code is shown, the statement is supported by all BASICs.

<b>ALLOCATE [C]</b>	Allocates file buffers
<b>BKOFF [C]</b>	Disable BREAK key
<b>BKON [C]</b>	Enable BREAK key
<b>CALL [I6]</b>	Access user routine
<b>CHAIN [I6]</b>	Invoke/merge subsequent program
<b>CLEAR</b>	Set aside bytes for string storage; Zero / clear variables
<b>CLOSE</b>	Closes file buffer(s)
<b>CLS</b>	Clear screen, home cursor
<b>CMD [I5]</b>	Invoke extra commands; exit to DOS
<b>COMMAND [C]</b>	Mechanism to define start of user command
<b>COMMON [I6]</b>	Passes selected variables to a chained program
<b>COMPL [C]</b>	Complement graphics pixel at (x,y); if pixel SET then RESET it, otherwise SET it
<b>CONT [I]</b>	Used to continue a STOPped program
<b>DATA</b>	Define a list of data

<b>DEC [C]</b>	Decrement integer variable by one
<b>DEFDBL</b>	Variables included in the list will default to double precision if their types are otherwise unspecified
<b>DEFFN</b>	Single line user defined function
<b>DEFINT</b>	Same as DEFDBL, except causes a default to integer
<b>DEFSNG</b>	Same as DEFDBL, except causes a default to single precision
<b>DEFSTR</b>	Same as DEFDBL, except causes a default to string type
<b>DEFUSR [I]</b>	Establish linkage to user machine language routine
<b>DIM</b>	Dimension specified arrays
<b>DOWN [C]</b>	Scroll entire screen down by one line
<b>DRAW [C]</b>	Using integer array as controller to set, reset, or complement turtle graphics on screen
<b>ELSE</b>	Defines default branch location if "IF" expression false
<b>END</b>	Stop program execution
<b>ENDCOM [C]</b>	Specify end of user defined command definition
<b>ENDFUNC [C]</b>	Specify end of multi-line user defined function definition
<b>ENDIF</b>	Terminate "IF" block
<b>ERASE [I6]</b>	Delete an array

<b>ERROR</b>	Force an “artificial” run-time error of error code <i>exp8</i>
<b>FIELD</b>	Fields file buffer into blocks of strings
<b>FOR</b>	Start a “FOR ... NEXT” loop construct
<b>FUNCTION [C]</b>	Start multi-line user function definition
<b>GET</b>	Reads one record from a file into its buffer
<b>GOSUB</b>	Call subroutine at program line or label
<b>GOTO</b>	Branch to program line or label
<b>IF</b>	Define beginning of conditional execution program block
<b>INC [C]</b>	Increments integer variable by one
<b>INPUT</b>	Accept user keyboard input for variable values
<b>INPUT#</b>	Assign variable(s) information read sequentially from specified disk file
<b>INPUT\$ [I6]</b>	Input a string of characters without echo
<b>INPUT@ [I5]</b>	Fielded screen input
<b>INVERT [C]</b>	Inverts all graphics on the screen
<b>JNAME [C]</b>	Define line label
<b>KEY [C]</b>	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
<b>KILL [I]</b>	Delete specified filespec from disk

<b>LEFT [C]</b>	Scroll entire screen left one character
<b>LET</b>	Set variable equal to algebraic or string expression
<b>LINEINPUT</b>	Assign string variable from verbatim keyboard input without default “?” prompt
<b>LINEINPUT#</b>	Assign string variable from line of disk input
<b>LINESPAGE [C]</b>	Sets printed lines per page
<b>LMARGIN [C]</b>	Sets left hand margin
<b>LOAD</b>	Loads the program file specified by filespec\$
<b>LPRINT</b>	Send list of information to printer
<b>LSET</b>	Sets var\$ = exp\$, with left justification
<b>MERGE [I]</b>	Merge ASCII source program
<b>MID\$()=</b>	Overlay var\$ starting at position <i>exp1</i> with <i>exp\$</i> for a maximum of <i>exp2</i> characters
<b>NEXT</b>	Define end of “FOR ... NEXT” loop
<b>ON</b>	Using expression, jumps to specified # in list
<b>ON BREAK [C]</b>	Causes branch to specified line or label if BREAK key hit and break scan active (BKON mode)
<b>ON ERROR</b>	Causes a branch to the specified line or label if (run-time) error occurs
<b>OPEN</b>	Opens a file using the specified buffer #
<b>OPTION BASE [I6]</b>	Sets base index of arrays
<b>OUT</b>	Send <i>exp2</i> out to port <i>exp1</i>

<b>PAGELEN [C]</b>	Sets physical lines per page
<b>PAINT [C]</b>	Color a bounded shape
<b>PLOT [C]</b>	Plots a line or a box on the screen
<b>POKE</b>	Load memory location <i>exp1</i> with <i>exp2</i>
<b>POP [C]</b>	Delete last GOSUB
<b>POSFIL [C]</b>	Position to specified point in sequential file. Functional with both "O" and "I" type files
<b>PRINT</b>	Output list of information to screen
<b>PRINT#</b>	Output list of information to disk file
<b>PUT</b>	Writes the buffer contents to a file
<b>PZONE [C]</b>	Define or clear printer TAB stops
<b>RANDOM</b>	Initializes the random number generator
<b>RDGOTO [C]</b>	Positions DATA pointer to specified line number or label
<b>READ</b>	Reads a list of variables from DATA statements
<b>REM or '</b>	Define a remark
<b>REPEAT [C]</b>	Define beginning of REPEAT/UNTIL construct
<b>RESET [I5,C]</b>	Turn off a graphics point at x,y
<b>RESTORE</b>	Restores "DATA" pointer to data statement
<b>RESUME</b>	Used at the conclusion of an error trapping routine to jump to the specified line number
<b>RETURN</b>	Return from subroutine

<b>RIGHT [C]</b>	Scroll entire screen right one character
<b>RMARGIN [C]</b>	Set printer right margin
<b>RSET</b>	Sets <i>var\$</i> = <i>exp\$</i> , with right justification
<b>ROT [C]</b>	Set rotation offset (in 256 degree units) for subsequent DRAW statement executions
<b>RUN</b>	Loads and executes the program specified
<b>SCALE [C]</b>	Set scalar line multiplier (in 1/256 units) for subsequent DRAWS.
<b>SCLEAR [C]</b>	Initialization command for "SORT". Use before any KEYing/TAGing done
<b>SET [I5,C]</b>	Set graphics point at x,y
<b>SETEOF [I5,C]</b>	Set random file's end-of-file
<b>SORT [C]</b>	SORT of KEYed and TAGed arrays.
<b>SOUND [I6,C6]</b>	Generate a tone
<b>STOP</b>	Stops execution of the program and prints source line number if available
<b>SWAP [I6,C]</b>	Swap contents of two like-typed variables
<b>SYSTEM</b>	Invoke a DOS command string; exit to DOS
<b>SZONE [C]</b>	Define or clear screen TAB stops
<b>SWAP [I6,C]</b>	Exchanges <i>var1</i> and <i>var2</i> 's values
<b>TAG [C]</b>	TAG array for SORTing purposes
<b>THEN</b>	Defines branch location for true "IF" expression
<b>TROFF</b>	Turn program trace OFF

TRON	Turn program trace ON
UNTIL [C]	Defines end of "REPEAT ... UNTIL" construct. Program execution branches back to last executed "REPEAT" if <i>exp</i> <> 0
UP [C]	Scroll entire screen up by one line ("conventional" scroll)
USR [I]	Invoke machine language routine
WAIT [I6]	Pause until specified input from port
WHILE [I6]	Initiate a looping construct
WEND [I6]	Terminate "WHILE" construct
WIDTH [I6]	Establish width of print line
WPOKE [C]	Does two byte poke of <i>exp</i> at <i>addr</i>
WRITE [I6]	Output comma terminated data
XFIELD [C]	Field "X"-type files

### String functions

Note: The codes enclosed in square brackets indicate the BASIC supporting the particular function. These codes are: I=Interpreter BASIC, I5=DOS 5 Interpreter BASIC, I6=DOS 6 Interpreter BASIC, C=Compiler BASIC, C5=DOS 5 Compiler BASIC, C6=DOS 6 Compiler BASIC. When no code is shown, the function is supported by all BASICs.

BIN\$ [C]	Convert <i>exp</i> to 16 digit base 2 representation
CHR\$	Convert <i>exp8</i> to one byte string
DATE\$ [I6,C]	Obtain system date
ERRS\$ [I6]	Obtain DOS error string

HEX\$ [I6,C]	Convert <i>exp16</i> to four digit hexadecimal representation
INKEY\$	Last key pressed on keyboard
LEFT\$	Return <i>exp</i> left most characters in <i>exp\$</i>
MID\$	Return substring of string
MKD\$	Convert <i>exp</i> to eight-byte string representing a double precision Floating Point number
MKI\$	Convert <i>exp</i> to two-byte string representing an integer number
MKS\$	Convert <i>exp</i> to four-byte string representing a single precision Floating Point number
OCT\$ [I6]	Convert <i>exp</i> to octal digit string
RIGHT\$	Return <i>exp</i> right most characters in <i>exp\$</i>
SPACE\$ [I6]	Obtain string of spaces
STR\$	Return ASCII DECIMAL equivalent of <i>exp</i>
STRING\$	Return <i>exp1</i> long string of <i>exp2</i> characters
TIME\$	Obtain system time (and date)
USING [C]	Return string using varlist, formatting determined by <i>format\$</i> . Takes the place of the "PRINT USING ..." feature in interpretive BASIC. Performs equivalently
WINKEY\$ [C]	Wait for key and then return as one-char string

### Numeric functions

Note: The codes enclosed in square brackets indicate the BASIC supporting the particular function. These codes are: I=Interpreter BASIC,



I5=DOS 5 Interpreter BASIC, I6=DOS 6 Interpreter BASIC, C=Compiler BASIC, C5=DOS 5 Compiler BASIC, C6=DOS 6 Compiler BASIC. When no code is shown, the function is supported by all BASICs.

<b>&amp;Bd0...d15</b>	Accept digits in base 2 representation
<b>&amp;Hdddd</b>	Accept digits in base 16 representation
<b>&amp;Odddd</b>	Accept digits in base 8 representation
<b>ABS</b>	Returns the absolute value of the expression
<b>ADDRA [C]</b>	Absolute memory address of "label" or line#
<b>ASC</b>	Returns the ASCII numeric code of the first byte of the string expression
<b>ATN</b>	Returns the arctangent (in radians) of the expression
<b>CDBL</b>	Converts expression to a double precision value
<b>CINT</b>	Converts expression to an integer value
<b>COS</b>	Returns the radian cosine of expression
<b>CSNG</b>	Converts expression to a single precision value
<b>CURLOC [C]</b>	Current cursor position (0-1023)
<b>CVD</b>	Directly copies eight-byte string to a double precision numeric expression
<b>CVI</b>	Directly copies two-byte string to an integer expression
<b>CVS</b>	Directly copies four-byte string to a single precision expression
<b>EOF</b>	Returns "-1" if at end of specified sequential input file, "0" otherwise

<b>ERL</b>	Line number of the latest error
<b>ERR</b>	Code of the latest error
<b>EXISTS [C]</b>	Returns "-1" if <i>filespec\$</i> exists.
<b>EXP</b>	Returns the natural antilog of expression
<b>FIX</b>	Returns the integer value of the expression
<b>FRE</b>	Returns amount of free string space (or MEM if <i>exp</i> = 0)
<b>INP</b>	Returns eight bit value read from port <i>exp</i>
<b>INSTR</b>	Returns "0" if <i>exp1\$</i> does not contain <i>exp2\$</i> , else returns the position of <i>exp2\$</i> 's first occurrence in <i>exp1\$</i> .
<b>INT</b>	Return greatest integer less than <i>exp</i>
<b>LEN</b>	Length of <i>exp\$</i>
<b>LOC</b>	Returns last record accessed in specified random file
<b>LOF</b>	Returns number of records in specified file
<b>LOG</b>	Natural log of <i>exp</i>
<b>LPOS [I]</b>	Obtain print head relative position
<b>MEM</b>	Amount of free memory
<b>PEEK</b>	Eight bit contents of memory address <i>exp16</i>
<b>POINT [I5,C]</b>	Returns "-1" if specified point is SET
<b>POS</b>	Intra-line cursor position
<b>RND</b>	Returns a random number between 1 and <i>exp</i>

ROW	Obtain cursor row number
SGN	Signum function (1 if $\text{exp} > 0$ , 0 if $\text{exp} = 0$ , -1 if $\text{exp} < 0$ )
SIN	Returns radian sine of $\text{exp}$
SPC [I6]	Generate spaces in "PRINT"
SQR	Returns square root of $\text{exp}$
TAN	Returns radian tangent of $\text{exp}$
TYPE [C]	Returns variable type of $\text{exp}$
USR [I]	Invoke user routine and obtain result
VAL	Changes ASCII DECIMAL string to internal numeric binary storage format
VARPTR	Absolute memory location of the specified variable or array element
WPEEK [C]	Returns two byte contents $(\text{addr}) + 256(\text{addr}+1)$
array() [C]	Address of the DCB of the specified array.

**Numeric BINARY operators**

Algebraic Operators			
"↑"	A↑B	A to the Bth power	C5, I5
"^"	A^B	A to the Bth power	C6, I6
"*"	A*B	A multiplied by B	
"/"	A/B	A divided by B	
"+"	A+B	A plus B	
"-"	A-B	A minus B	

In the A^B and ↑ operators, if A is negative, B must be integer, otherwise an illegal function call error will prevail.

**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 ANDed with B	
"OR"	A OR B	A Logically ORed with B	
"XOR"	A XOR B	A logically XORed with B	I6,C
"NOT"	NOT A	1's Complement of A	
"EQV"	A EQV B	(A AND B) OR (NOT(A AND B))	I6
"IMP"	A IMP B	A AND NOT B	I6

**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\$ ordered less than B\$
">"	A\$>B\$	A\$ ordered greater than B\$
"<="	A\$<=B\$	A\$ ordered less than or equal to B\$
">="	A\$>=B\$	A\$ ordered greater than or equal to B\$
"<>"	A\$<>B\$	A\$ is not equal to B\$

**Variable storage format**

The following information describes the control block of arrays and the data storage format of the four supported variable types. For scalars, a

pointer to the data area is returned by the "VARPTR" function. For arrays, the Compiler BASIC "VARPTR" function or its array counterpoint, "arrayname()" returns a pointer to the data control block. The Interpreter BASIC VARPTR function returns a pointer to the specified array element; the data control block always precedes the first element of the array.

<b>Numeric Storage Format</b>	
<b>Integer Storage Format</b>	<b>Description of contents</b>
LSB HSB	Value of the integer, 2-bytes
<b>Single Precision Format</b>	<b>Description of contents</b>
LSB MSB HSB EXP	Value of the single, 4-bytes
<b>Double Precision Format</b>	<b>Description of contents</b>
LSB MSB ... MSB HSB EXP	Value of the double, 8-bytes

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

<b>Array Control Blocks</b>	
<b>Compiler BASIC</b>	
<b>Array Data Control Block</b>	<b>Description of contents</b>
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

**Interpreter BASIC**

<b>Array Data Control Block</b>	<b>Description of contents</b>
DCB+0	Array type: 02=integer, 04=single prec, 03=string, 08=double precision.
DCB+1&2	ASCII-coded array name
DCB+3&4	Number of data elements
DCB+5	Number of dimensions
DCB+6&7 on up	Size of each dimension

**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 of Compiler BASIC 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). Supplied functions of Interpreter BASIC support only single precision arguments and results.

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

## **File buffer allocation**

For each Interpreter BASIC file buffer designated via "F=" parameter, 564 bytes of memory will be provided for DOS 6 and 290 bytes (546 if Block is ON) for DOS 5. This memory is utilized as follows:

<b>Buffer offset</b>	<b>Intended use</b>
0	File type: 1="I", 2="O", 3="R" ("E" is converted to "O")
1	Unused
2-33 (2-51)	System's File Control Block
34-289 (52-307)	File's 256-byte I/O buffer
290-545 (308-563)	File's user record buffer

Note: Numbers in parentheses refer to DOS 6 BASIC

For each Compiler BASIC 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

## **Compiler BASIC 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.             | 014 - Convert the word on the stack to an integer number.           |
| 001 - POP stacked operands and set up for math routines.      | 015 - Interface to the @DATE and @TIME DOS functions.               |
| 002 - Floating point addition.                                | 016 - Load the following string literal onto the string stack.      |
| 003 - Print evaluated expression.                             | 017 - This performs the NEXT command of BASIC.                      |
| 004 - POP operand and place in the math memory accumulator.   | 018 - Specified variable read from current DATA statement.          |
| 005 - Floating point multiplication.                          | 019 - The two topmost strings on the string stack are concatenated. |
| 006 - Floating point division.                                | 020 - "MID\$(exp\$, A, B)".   |
| 007 - Floating point subtraction.                             | 021 - Load the following string variable onto the string stack.     |
| 008 - Arithmetic OR (integers).                               | 022 - Handles LET S1\$=S2\$.  |
| 009 - Arithmetic AND (integers).                              | 023 - Handles "ON exp GOTO/GOSUB".                                  |
| 010 - Compare the last two stacked operands for less than.    | 024 - Allocate tempy string space.                                  |
| 011 - Compare the last two stacked operands for greater than. | 025 - Check the stack pointer for SP < (PRGTOP)+256.                |
| 012 - Compare the last two stacked operands for equality.     | 026 - Test exp1\$ <> exp2\$.  |
| 013 - Arithmetic XOR (integers).                              | 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 ..."
- 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 - Re-initializes 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 buffer expression.
- 085 - Handles "MID\$() = exp\$".
- 086 - "POSFIL" assessor 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 - Used by END.
- 168 - "ALLOCATE" executor.
- 169 - "FIELD" executor.
- 170 - "IF" executor.
- 171 - "XFIELD" executor.
- 172 - RESUME line#.
- 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 - Z80 routine call.
- 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" & "FOR".
- 230 - Handles "POKE exp1,exp2".
- 231 - Handles "WPOKE  
exp1,exp2".
- 232 - Begin execution of Z-80  
code.
- 233-255 - unused.

### Compiler BASIC Z80 Assembler Introduction

Compiler BASIC, 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.

### Z80 Source Code Inclusion in Programs

Z80 assembly language can be inserted at any point in the source program. The "Z80-MODE" Compiler Directive 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 colons, ":"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 "SCREEN"
70 Z80-MODE
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 defines a label, 'SCREEN'.

Line 70 switches the compilation context to Z80 AL

Lines 80-90 define the Z80 subroutine itself.

Line 100 switches the compilation context back to BASIC.

### Access of BASIC variables and line numbers

The previous sample program illustrates that you can reference a BASIC variable using the syntax,

**&(varname)**

You can also reference the address of a BASIC line number (an actual line-numbered line) by the syntax,

**&(#line\_number)**

For example, the Z80 code,

**LD HL,&(SCALAR)**

loads the address of the BASIC variable scalar into the HL register, whereas the Z80 code,

**LD HL,(&(SCALAR))**

is an indirect 16-bit load of the memory contents of the variable into the HL register. If SCALAR was an integer, register HL would then contain the integer value. The reference material for COMMAND and FUNCTION contain more examples of variable access.

### 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".

### Binary Operators

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

**Algebraic Operators**

"<"	exp1 < exp2	Exp1 shifted left 'exp2' time
">"	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 algebraic 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

## Operand Bases

The following table describes the allowable numeric operand bases:

<b>No suffix:</b>	Base 10 = Decimal = Regular number
<b>“V” suffix:</b>	Base 2 = Binary ex: 1011V = 11 decimal
<b>“H” suffix:</b>	Base 16 = Hexadecimal ex: 4000H = 16384 decimal
<b>“O” suffix:</b>	Base 8 = Octal ex: 500 = 40 decimal

## Non-standard Z80 Instructions

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

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



## **Assembler Pseudo-OPs**

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 exp16 <,exp16,...>**  
Define word(s)

**DEFS exp16**  
Leave 'exp16' bytes untouched

**DEFF exp16 <,exp8>**  
Fill 'exp16' bytes with 00H. Optionally fill with  
'exp8' if given

**ORG exp16 and DISORG**  
Start a separate machine language load block with  
starting load address given by exp16. 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

**Compiler BASIC error codes****Compile-time Errors**

<b>Error Code</b>	<b>Meaning</b>
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
<hr/>	
192	(Z80) Expression error
193	(Z80) Relative branch out of range
194	(Z80) Operand field OVERFLOW

**RUNTIME 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	Access does not match file mode
110	File already opened
112	Field overflow (XFIELD)
114	"X" record number overflow
122	Disk space full
124	End of file encountered
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	Bad record length for access specified
178	Attempt to open file with different LRL
180	Buffer not open
241	SORT attempted without sort keys given
242	Too many sort keys or tags
254	Bad file mode (not "I", "O" or "R")

### Interpreter error codes

Incorporated in non-disk BASIC (ROM, where applicable) are various error messages and error codes. These error codes are provided for the user so that certain types of errors may be “trapped” for, and the execution of the BASIC program in question will not be interrupted. You may determine the exact nature of an error by utilizing the ERR and ERL commands.

BASIC will have in its error dictionary, disk error codes, along with the error codes classically found in ROM BASIC. The error dictionary for DOS 5 Interpreter BASIC is contained in the file BASIC/OV3. For this reason, BASIC/OV3 must always be present on a disk in the system when programming in BASIC.

This part of the manual will list the error codes and messages, and will include a brief description of each error. The user should realize that the descriptions given for each error are not all inclusive. That is to say, the example circumstances given for a particular error may not encompass all circumstances which could generate the error in question.

Before we begin giving these error codes, a few general points should be made. BASIC's error dictionary is not as large as the error dictionary found in DOS. For this reason, several different types of disk related errors may produce the same BASIC error message. To pinpoint the exact nature of a disk related error, it may be beneficial to determine the DOS interpretation of an error. After a disk related error occurs, you may determine the associated DOS error message by performing a CMD"E" in DOS version 5 BASIC, or use the ERRS\$ function in DOS version 6 BASIC. This may be useful when, for instance, you get the BASIC error message “Disk I/O Error”, as several different occurrences may cause this type of error. For more information, refer to CMD"E" and ERRS\$.

---

## Error Codes

---

The following tables show Interpreter BASIC error codes. Note that all error codes given here will be the value returned by the ERR command.

Error	Code	Meaning
15	16	
0	1	Next without For
2	2	Syntax error
4	3	Return without GOSUB
6	4	Out of Data
8	5	Illegal Function Call
10	6	Overflow
12	7	Out of memory
14	8	Undefined line number
16	9	Subscript out of range
18	10	Re-dimensioned array; Duplicate Definition
20	11	Division by zero
22	12	Illegal direct
24	13	Type mismatch
26	14	Out of string space
28	15	String too long
30	16	String formula too complex
32	17	Can't continue
	18	Undefined user function
34	19	No RESUME
36	20	RESUME without error
38	21	Unprintable error
40	22	Missing operand
42		Bad file data
	23	Line buffer overflow
	26	FOR without NEXT
	29	WHILE without WEND
	30	WEND without WHILE

Error Code		Disk Errors
15	16	
100	50	Field overflow
102	51	Internal error
104	52	Bad file number
106	53	File not found
108	54	Bad file mode
110	55	File already open
114	57	Disk I/O error; Device I/O error
116	58	File already exists
122	61	Disk full
124	62	Input past end
126	63	Bad record number
128	64	Bad file name
132	66	Direct statement in file
134	67	Too many files
136	68	Disk write protected
138	69	File access denied
140		Blocked file error
142	70	System command aborted
144		Protection has cleared memory

## **Error Definitions**

**Bad File Mode** indicates that a file is being accessed improperly. This may occur when, for instance, you try to access a file opened as a random file in a sequential manner (i.e. issue an INPUT# command after opening a file in the random mode).

**Bad File Name** will be generated when the file specified in an OPEN, SAVE, LOAD, RUN or MERGE command does not conform to the rules governing valid DOS filespecs.

**Bad File Number** will occur when a file is opened using an illegal buffer number (a buffer number greater than the total number of files specified when entering BASIC), or fielding a buffer which does not correspond to an open random file.

**Bad Record Number** will be issued when record number 0 (or some other illegal record number) is accessed in a random file.

Blocked File Error will occur if you attempt to OPEN a random file with an LRL of other than 256 after entering BASIC and specifying the parameter BLK=OFF.

Can't continue results when a CONT statement is entered at a point not within the scope of the CONT statement.

Direct Statement in File will be generated when a LOAD is performed of a file that is not a BASIC program (usually when a LOAD of a data file is attempted). This type of error will also be generated when a BASIC program which was saved in ASCII is loaded, and a line in the program exceeds 240 characters in length.

Disk Full will indicate that all of the free space on a disk has been consumed. In some cases, the occurrence of a disk becoming full (i.e. all of the disk space being consumed) may generate a Disk Write Protected error.

Disk I/O Error or Device I/O error will occur when an input from or an output to a disk file is unsuccessful. A typical DOS error message which is associated with the it is a Parity Error.

Disk Write Protected usually indicates that a write has been attempted to a write protected disk. Other types of errors may also generate a Disk Write Protected error. If the disk in question is not write protected, use CMD"E" to determine the exact error.

Division by zero will be generated when a division calculation has a zero divisor.

Field Overflow indicates that the number of bytes fielded for a random file exceeds the record length of the file (as specified in the OPEN statement).

File Access Denied may be generated when a password protected file (either a data file or a program file) is referenced using an incorrect password.

File Already Exists will be generated when using an OPEN"xN" command if the file already exists, or a NAME statement target exists.

File Already Open will be generated when you try to OPEN a file using a buffer that corresponds to an already open file. Note that no error will be

generated if the same file is in an open state using two different buffers at the same time (This practice is NOT advised).

File Not Found indicates that the file being referenced does not exist. This error may occur after an OPEN"I", OPEN"EO", OPEN"OO", OPEN"RO", LOAD or RUN command has been issued.

FOR without NEXT occurs when a FOR statement has no corresponding NEXT statement.

Illegal direct is issued when a command is entered in *immediate* mode which is improper for that mode.

Illegal function call indicates any of a number of errors relating to bad parameters of a function (e.g. a negative argument to LOG or SQR).

Input Past End applies only to sequential files opened for input, and will occur when a read of the file is attempted after all data in the file has been input.

Internal Error will occur when the error in question cannot be interpreted. One way it may be generated is to issue a CMD"L" command, and the file to be loaded is not found.

Line buffer overflow will occur when an INPUT exceeds the size of the input buffer.

Missing operand will be generated when a required operand of an operator is omitted, e.g. "IF A AND".

NEXT without FOR will be generated when a NEXT statement is encountered without a corresponding FOR statement active.

No RESUME will be issued when the code block of an error-trapping ON ERROR GOTO routine does not have a RESUME statement.

Out of data will be generated when a READ statement is encountered with either a missing DATA statement or no unread data remains in a DATA statement.



**Out of memory** occurs when a program along with its run-time data space usage exceeds the memory available to it.

**Out of string space** is generated insufficient free memory exists to allocate the designated string.

**Overflow** results when a calculation exceeds the largest representative number, regardless of sign.

**Protection Has Cleared Memory** will be generated if an attempt is made to illegally access an EXECute only program without using the proper password. The program and variables will be cleared from memory.

**Re-dimensioned array or Duplicate definition** is generated if a DIM statement is encountered for an array already dimensioned by a preceding DIM statement.

**RESUME without error** will be generated when a RESUME statement is encountered outside of a pending ON ERROR GOTO error trap.

**RETURN without GOSUB** will be generated when a RETURN statement is encountered without a corresponding GOSUB statement active.

**String formula too complex** can occur in some very complex string expressions which require that the expression be partitioned into smaller expressions.

**String too long** occurs when the result of a string expression would exceed the maximum character length limit for a string.

**Subscript out of range** will be generated when an array subscript which exceeds the bounds declared in the DIM statement is encountered.

**Syntax error** indicates a programming construct error in the specified offending line.

**System Command Aborted** will occur if a DOS command called by the CMD"command" or SYSTEM"command" function is manually aborted.

**Too Many Files** will occur when an attempt is made to add another extent to a file when all directory entries have been used. This type of error will be very uncommon.

**Type mismatch** is generated when a string variable or expression is used when a numeric variable or expression is required, or vice versa.

**Undefined line number** occurs when the target line number of a GOTO or GOSUB, for example, is not present in your program.

**Undefined user function** will be generated when a USRn is invoked without a corresponding DEFUSR definition of "n".

**Unprintable error** will occur if an error other than one detailed in this list is encountered.

**WEND without WHILE** will be generated when a WEND statement is encountered without a corresponding WHILE statement.

**WHILE without WEND** will be generated when a WHILE statement starting a looping block has no closing WEND statement.

## Index

	*IF, 149
	*INCLUDE, 51,55,246,253
	/
!	/BAS extension, 5
!, 3,27,29,30,101,126,179,208, 215	@
#	@CALADR, 247
	@CKDRV, 276
	@CMNDI, 276
	@CMNDR, 276
	@EXIT, 258
#, 3,27,101,177	@pos, 140,155,204
#, in USING, 214	@VDCIL, 275,277
\$	\
\$, 3,27,101	\, in USING, 215
\$\$, in USING, 214	^
%	^, 257,301,308
	^^^, in USING, 214
%, 3,27,101,178,208,245	6
%, in USING, 215	64180, 139,193,282
&	A
&, 254,312	abbreviated commands, 19
&, in USING, 215	ABS, 62,299,308
&B, 67,299	absolute value, 62
&H, 133,272,275,277,299	addition, 301
&O, 180,299	ADDRA, 63,299
*	ALLOCATE,
	64,119,163,170,177,185,190
*, 21	,203,246,291,308
**\$, in USING, 214	AND, 180,275,282,302,306,313
** , in USING, 214	
*ENDIF, 149	
*GET, 51,55	

apostrophe as remark, 223  
appending lines, 44  
Appending support subs, 50  
arc-tangent, 66  
array allocation, 12  
array control block, 303,304  
array dimensions, specifying,  
    105  
array subscript, 192  
array, sorting, 82,192,250,263  
arrays, eliminating, 112  
AS, 31,120,163,168,177,190,246  
ASC, 65,138,146,299,308  
ascending, 250  
ASCII file, 42,45  
ASCII format, 16,72  
ASCII program, 171  
ASCII value, 138  
assembler, 12  
assembler Pseudo-Ops, 315  
assembly language, 58,311  
assign a value to a variable, 154  
ATN, 66,299,308  
AUTO, 19,20,44  
automatic input mode, 20

## **B**

Bad File Mode, 320  
Bad File Name, 320  
Bad File Number, 320  
Bad parameters, 29  
Bad Record Number, 320  
BASIC compiler, 10  
BASIC line numbers, 34,37  
BASIC/CMD, 4  
BASIC/HLP, 4  
BASIC/OV1, 4  
BASIC/OV2, 4,26,32  
BASIC/OV3, 4,26,82,318  
BASIC/OV4, 5

BC/CMD, 10  
BIN\$, 13,68,181,297,307  
binary digits, 68  
binary number, 67  
BKOFF, 69,183,291  
BKON, 69,183,291  
Blocked File Error, 321  
blocked file mode, 6,8  
box, 197  
branching, 132,182  
BREAK, 44,52  
BREAK key,  
    12,13,57,58,81,90,138,183,2  
    91,309  
BREAK key control, 69  
BREF/CMD, 31  
BRKOFF, 308  
BRKON, 308  
BRL, 183,308  
BSORT, 192,263  
buffer address, obtaining, 280  
buffer partitioning, 119

## **C**

CALL, 70,291  
calling subroutines, 130  
Can't continue, 321  
Caps lock, 276  
cassette, 262  
cassette baud rate, 6  
cassette tape I/O, 9,83  
CDBL, 71,299,308  
CED/CMD, 10  
CHAIN, 72,87,239,291  
CHR\$,  
    63,74,146,224,238,252,277,  
    297,307  
CINT, 75,299,308,309  
CLEAR, 76,78,190,240,252,291  
CLOAD, 18

- clock display, 83
  - CLOSE,
    - 78,121,177,190,246,291,307
  - closing open files, 26
  - CLS,
    - 79,109,124,242,275,291,308
  - CMD, 80,239,291
  - CMD"A", 78,81
  - CMD"B", 81
  - CMD"D", 81
  - CMD"E", 82,318
  - CMD"I", 78,82,110
  - CMD"L", 82,161
  - CMD"N", 29,82
  - CMD"O", 82
  - CMD"P", 83
  - CMD"R", 9,83,263
  - CMD"S", 78,83
  - CMD"T", 9,83,263
  - CMD"V", 33
  - CMD"X", 31,32,83
  - CMDFILE, 263
  - color, 195
  - comma, explicit, 207
  - COMMAND, 225,247,253,291
  - Command-line compiling, 53
  - commands, user-defined, 48
  - COMMON, 72,87,291
  - comparisons, 302
  - compiler directives, 12
  - compiling a source program, 50
  - COMPL,
    - 89,109,124,197,227,244,291
  - complement a pixel, 89
  - conditional execution, 135
  - CONT, 90,110,258,291
  - continuing a program, 90
  - convert to double precision,
    - 70,71,94
  - convert to integer, 75,95
  - convert to single precision,
    - 92,96
  - converting numeric to hex
    - string, 134
  - copy, 19,26,36
  - Copyright, ii
  - COS, 91,124,265,299,304,308
  - cosine, 91
  - CPU port, 139,193,282
  - CREATE, 163
  - Creating new file, 50
  - cross reference, 10,31,46
  - cross reference, generating,
    - 58,83
  - CSAVE, 18
  - CSNG, 92,299,308
  - CURLOC, 93,237,299,308
  - cursor, 19,155,202,205,291
  - cursor position, 202
  - cursor position, obtaining, 93
  - cursor, obtaining position of,
    - 237
  - CVD, 94,176,299,307
  - CVI, 95,178,299,307
  - CVS, 96,179,299,307
  
  - D**
  
  - DATA,
    - 97,221,222,229,263,291,306
  - data list, positioning, 221,229
  - data list, reading, 222
  - DATE\$, 99,297
  - date, system, 266
  - debugger, 81
  - DEC, 100,292,308
  - DEF FN, 73
  - default file extension, 6,7
  - DEFB, 315
  - DEFDBL, 3,73,101,292
  - DEFF, 315
  - DEFFN, 102,126,292
  - defining new commands, 84
-

DEFINT,  
    3,73,101,109,236,240,274,2  
    92,311  
DEFM, 315  
DEFS, 315  
DEFSNG, 3,73,101,292  
DEFSTR, 3,73,101,240,292  
DEFUSR, 104,133,272,292  
DEFW, 315  
DELETE, 19,21  
    delete, 36  
    deleting the current program, 26  
    delimiter, 166,204  
    delimiter, PRINT#, 207  
    delimiter, string input, 210  
    descending, 250  
    Device I/O error, 321  
DIM,  
    105,109,112,150,192,231,23  
    6,252,263,274,292,307  
DIM statement, 12  
Direct Statement in File, 321  
Direct statement in file, 17  
directive prefix, 54  
directives in your source, 55  
directory of files, 44  
disk file, deleting, 24  
Disk Full, 321  
Disk I/O Error, 321  
Disk Write Protected, 321  
DISORG, 315  
division, 301  
Division by zero, 321  
DOS command, 262  
DOS command, invoking, 82  
DOS commands, invoking, 80  
DOS error, 51  
DOS error code, 52  
DOS error message, 82  
DOS error, obtaining, 116

double precision,  
    3,12,66,91,176,235,249,257,  
    268,304  
DOWN, 107,292,308  
DRAW, 108,236,243,292  
DUPI, 314  
duplicate a line, 26  
Duplicate definition, 323  
  
**E**  
  
EDIT, 2,19,22,37,78  
edit commands, 22  
edit string, 40  
editing Compiler BASIC, 34  
editing line numbers, 34  
Editing of Interpreter BASIC, 15  
editor, 10  
ELSE, 244,292  
encoded format, 16  
END,  
    44,52,90,110,182,201,246,2  
    58,292,307,308  
ENDCOM, 84,225,247,254,292  
ENDENC, 149  
ENDFUNC, 126,281,292  
ENDIF, 292  
EnhComp, 1  
ENLOOP, 149  
ENTER, logical, 40  
EOF, 111,299,307  
EQ, 313  
EQU, 55,149  
EQV, 302  
ERASE, 112,292  
ERL, 113,183,257,300,308,318  
ERR, 114,257,300,308,318  
ERROR, 36,115,184,293,307  
    error code, 114  
    error codes, 316,318  
    error control, 115,184,230

error message, 116  
error message, waiting for, 58  
ERRS\$, 116,297,318  
Execute only, 8  
EXISTS, 117,300,307  
exit, 262  
exit from BASIC, 6  
exit from the editor, 37  
exit to DOS, 110,291  
EXP, 118,164,300,304,308  
exponential, 118  
expression, 34  
expressions, 13  
extending files, 188

## **F**

FACTORIAL, 85,127  
FCB, 186  
Fibonacci, 98,221,229  
FIELD,  
    119,129,163,167,176,177,18  
    9,190,238,246,293,308  
field length, 121  
Field Overflow, 321  
Field overflow, 121  
fielding a record buffer, 289  
File access denied, 321  
File already exists, 188,321  
File already open, 321  
file buffer allocation, 305  
file closing, 78  
file control blocks, 64  
file input, 141,143,157  
File not found, 26,187,322  
file, checking if available, 117  
file, direct, 217  
file, random access, 238  
file, viewing, 45  
file, writing, 288  
Files, 5

files, direct, 189  
files, directory of, 44  
files, extending, 188  
files, list directed, 189  
files, maximum number of, 5  
files, opening, 185  
finding keywords, 27  
finding variables, 27  
FIX, 122,300,308  
FOR,  
    146,183,198,236,283,293,31  
    0  
FOR ... NEXT, 12,123,224  
FOR without NEXT, 322  
FORMS, 160,194,234,284  
forms control, 46  
FRE, 125,300,308  
free memory, 170  
FUNCTION, 126,281,293  
function, 61  
function name, 102  
functions, 13,47  
functions, multi-line, 126  
functions, user-defined, 102

## **G**

garbage collection, 154  
GET,  
    129,162,189,218,245,246,28  
    9,293,307  
GO, 37  
GOSUB,  
    130,201,231,275,293,308,31  
    1  
GOTO,  
    111,132,146,230,293,308  
graphics,  
    108,148,195,197,199,236,24  
    3,244,291,292,295,296,307  
graphics commands, 89

GT, 313

GTEQ, 313

## **H**

HEX\$, 13,134,181,298,307

hexadecimal number, 133

HIGH, 6,9

high level support code,  
suppressing, 58

HIGH\$, 5,53,76,80,276

HIGH-MODE,

13,54,86,128,149,247,254

HITAPE, 9

## **I**

IF, 293,308

IF...ENDIF, 55

IF...THEN...ELSE, 135

Illegal direct, 322

Illegal function call,

26,28,29,164,257,274,301,3  
22

IMP, 302

INC, 100,137,293,308

INJECT, 56

INKEY\$, 138,285,298,307

INP, 300,308

INPUT,

117,140,155,224,293,307

Input past end, 111,142,322

input variable list, 102

INPUT#,

141,157,187,203,210,293,30  
7

INPUT\$, 143,293

input, waiting for, 282

INPUT@, 144,293

inserting a load file, 56

INSTR, 145,300,307

INT, 147,300,308

integer, 3,12,178,268,304

integer function, greatest, 147

integer truncation, 122

integer, decrementing, 100

integer, incrementing, 137

interactive RUN mode, 59

Internal Error, 322

INVERT, 148,293,308

invoking BASIC, 5

invoking user commands, 84

## **J**

JNAME, 149,293

## **K**

KEY, 131,250,293,308

keyboard entry, waiting for, 285

keyboard input, 138,140,144,155

keyword, 27,28

KILL, 24,41,293,307

## **L**

label, 55,132

label table, 48

LEFT, 151,294,308

LEFT\$, 152,175,277,298

LEN, 153,280,300,307

length of names, 2

LET, 154,294,306

library, 10

Line buffer overflow, 322

line label, 130,149

line number,

21,27,28,132,171,308

line number address, 63

line number suppression, 57

line number table, 49



line number zero, 30  
line number, error, 113  
line numbers, access from  
    assembler, 312  
line numbers, adding, 35  
line numbers, BASIC, 37  
line numbers, deleting, 36  
line numbers, editor, 37  
line numbers, generated, 54  
line numbers, generating, 58  
line numbers, hiding, 35,36  
line numbers, removing, 35  
line numbers, showing, 35,36  
line range, 40  
line, last, 34  
line, top, 34  
LINEINPUT, 155,294,307  
LINEINPUT#, 157,187,294,307  
lines, displaying, 45  
lines, erasing, 73  
LINESPAGE, 159,294  
LINK, 56  
LIST, 2,8,19,25,42,56  
list, 44  
LIST OCATE, 19  
listing the source program, 56  
literal character, 214  
LLIST, 8,25,43  
LMARGIN, 160,294  
LOAD, 7,9,15,161,294,307  
loading a load file, 82  
loading a program, 15,239  
loading address, default, 53  
loading source text, 41  
LOC, 162,300,307  
local directives, 53,54  
LOF, 163,300,307  
LOG, 118,164,300,304,308  
logarithm, 164  
LOGO, 108  
loop index, 123,124  
looping, 224,283

LOW, 6,9  
LPOS, 165,300  
LPRINT,  
    13,165,166,204,256,264,284  
    ,294,308  
LPRINT USING, 213  
LSET,  
    119,163,167,176,177,178,19  
    0,238,246,294,307  
LT, 313  
LTEQ, 313

## **M**

machine language, 271  
machine-language routine, 70  
margin, printer, 160,234  
MEM, 125,170,266,300,308  
Mem, 5  
MEMORY, 271  
memory address, 63,280  
memory protection, 5  
memory space, available, 125  
memory usage, 45  
memory, altering the contents  
    of, 200  
memory, free, 170  
memory, peeking, 286  
memory, poking, 287  
memory, protecting, 53  
memory, viewing, 196  
MERGE, 7,15,17,171,294  
merging a program, 72  
message, compilation, 57  
messages, printing, 58  
Microsoft, 2  
MID\$,  
    152,173,175,233,238,298,30  
    6,307  
MID\$0=, 294  
Missing operand, 322

MKD\$, 94,168,176,298,307  
MKI\$, 95,163,168,178,298,307  
MKS\$, 96,168,179,298,307  
MOD, 313  
Move, 27  
move, 43  
multiplication, 301

## **N**

names, 2  
negative index, 123  
NEQ, 313  
NEW, 26,43  
NEXT, 198,230,283,294,306  
NEXT without FOR, 322  
NO, 53,57  
No RESUME, 322  
NOLIST, 56  
Nomenclature, 1  
NOPRT, 57  
NOT, 302,308  
Not a BASIC program, 32  
number, convert to string, 259  
number, random, 235  
numeric formats, USING, 214  
numeric function, 298  
NX, 57,69,183,267

## **O**

object file, suppressing, 53  
object program, suppressing, 57  
OCT\$, 181,298  
octal number, 180  
ON, 294,306  
ON ... GOSUB, 182  
ON ... GOTO, 182  
ON BREAK, 294  
ON BREAK GOTO, 69,183,257  
ON ERROR, 294

ON ERROR GOTO,  
    52,113,114,115,184,230  
OPEN, 64,78,169,185,294,307  
open files, 73,110,240,242  
OPEN"E", 141,206  
OPEN"I", 111,141,206  
OPEN"O", 141,188,191,203,206  
OPEN"R", 163,177,190,246  
OPEN"X", 190,290  
opening existing files, 188  
opening new files, 188  
opening random access files,  
    189  
opening sequential files, 187  
operand bases, 314  
operators, binary assembler, 312  
operators, boolean, 302  
operators, boolean assembler,  
    313  
operators, logical, 302  
operators, numeric, 301  
operators, string, 302  
OPTION BASE,  
    73,105,112,192,263,294  
OR, 85,128,275,302,306,313  
ORG, 315  
OUT, 193,294,308  
Out of data, 322  
Out of memory,  
    27,32,81,240,323  
Out of memory error, 7  
Out of string space, 240,323  
output, redirecting, 216  
Overflow, 323  
overflow, 118

## **P**

page length, 194  
PAGELEN, 194,234,295  
PAINT, 195,295

- PEEK, 63,196,275,286,300,308
- PLOT, 195,197,295
- POINT, 199,227,300
- POKE, 200,275,295
- POP, 130,201,231,295,308
- port, 193
- port input, 282
- port, input from, 139
- POS, 202,300,308
- POSFIL, 187,203,295,307
- precision of numbers, 304
- prime number, 221,229
- PRINT,
  - 13,57,204,256,264,284,288,295
- print data, 166
- print position, 165
- PRINT USING, 213,270
- print width, 284
- print zones, 219
- PRINT#,
  - 141,187,203,206,256,295,307
- PRINT# USING, 213
- PRINT@, 93,243,308,311
- printed lines per page, 159
- printer, 165,194,219,234
- printer listing, suppressing, 57
- printer output width, 31
- printer status, 83
- printing lines, 43
- printing your program, 25
- program branching, 132
- program debugging, 267
- program invoking, 239
- Program Not Found, 30,32
- protected mode, 32
- Protection has cleared memory,
  - 8,323
- PRT, 58
- PUT,
  - 129,163,169,177,189,190,217,295,307
- PZONE,
  - 166,205,219,264,295,308
- R
- radians, 91,249,265
- RAM, 200,287
- RANDOM,
  - 89,220,228,235,295,308
- random, 224
- random access file,
  - 119,162,163,167,178
- random access file, reading, 129
- random access file, writing, 217
- random access files, truncating, 245
- random number, 220,235
- random number, seeding, 220
- Random Record Access, 186
- RDGOTO, 97,203,221,229,295
- Re-dimensioned array, 323
- re-entering BASIC, 6
- READ, 97,203,222,263,295,308
- README/TXT, 1
- record length, 185
- record number, 129,162,163,217
- REF/CMD, 10,46,58
- references, finding, 27
- REM, 223,276,295
- remark, entering a, 223
- REMOVE, 24
- renaming a disk file, 26
- RENUM, 2,29
- renumbering, 82
- renumbering a BASIC program, 29
- renumbering BASIC lines, 43
- REPEAT, 283,295,310

REPEAT ... UNTIL, 124,224

Replacing existing file, 50

replacing lines, 44

reserved word, 2

RESET,

89,109,148,197,227,244,295

RESTORE, 97,221,229,295

RESUME, 184,230,295,308

RESUME without error, 323

RET, 70,308

RETURN,

84,126,130,201,231,247,275  
,295,308

return to DOS, 37,81

RETURN without GOSUB, 323

RIGHT, 232,296,308

right justified, 238

RIGHT\$, 175,233,298

RMARGIN, 234,296

RND,

130,220,224,228,231,235,25  
2,300,308

ROT, 236,243,296,308

rotated figures, 108

ROW, 237,301,308

royalty payments, 59

RSET,

119,167,176,178,238,296,30  
7

RUN,

7,8,15,44,78,165,239,262,29  
6,307

Runtime errors, 52

## S

S/CMD, 10,59

SAVE, 7,9,16

saving space, 59

saving your program, 45

SCALE, 243,296,308

scientific notation, 214

SCLEAR, 131,250,296,308

screen print, 81

screen print zones, 264

scroll, 107,151,232,269,292,296

Search, 28

search and replace, 46

searching text, 37,45

Sequential Access, 186

sequential access file, 288

sequential file input, 141,157

sequential file, positioning, 203

sequential file, writing, 208

sequential file, writing strings,  
209

SET,

89,109,124,148,197,227,244  
,296

SETEOF, 245,296

SGN, 89,228,248,301,308

sign, 248

SIN, 124,265,301,308

sine, 249

single precision,

3,12,66,91,118,179,235,249,  
257,268,304

Single Stepping, 8

SORT, 131,250,296,307

sorting a string array, 82

SOUND, 253,296

sound generation, 253

SPACE characters, 256

SPACE\$, 255,256,298

SPC, 256,301

SQR, 257,301,308

square root, 257

stack, 76

stack, popping, 201

statements, 13,61

STEP, 124,198,236,264

STOP, 44,52,90,258,296,307

storing data, 97

storing programs, 16  
STR\$, 176,179,259,298,307  
string, 3,12,268  
string control block, 303  
string formats, USING, 215  
String formula too complex, 323  
string function,  
    152,173,175,233,255,259,26  
    0,266,270,297  
string input, 143,144  
string length, 3,153  
string length limit, 12  
String not found, 37  
string operators, 302  
string search, 145  
string to integer, 65  
String too long, 323  
string value, 279  
STRING\$, 13,190,260,284,298  
string, octal, 181  
string, one-character, 74  
sub-string, 145,152,175,233  
subroutine, 271,306  
Subscript out of range, 323  
subtraction, 301  
support routines, 51  
SUPPORT/DAT, 10,56,306  
SVC, 274  
SWAP, 261,296,308  
symbol table, 47,48  
Syntax error, 323  
SYSTEM,  
    80,163,190,246,262,296,307  
System Command Aborted,  
    80,323  
system date, 99  
system time, 266  
SZONE, 166,205,264,296,308

**T**

TAB, 166,204,264,274,307  
tab positions, 264  
TAB positions, setting, 219  
TAG, 131,250,296,308  
TAN, 265,301,308  
tangent, 265  
TEMP/BAS, 11,34,44,51  
TEMP/CMD, 44,51,59  
terminate your program,  
    110,258,262  
terminating AUTO, 21  
terminator, 256  
terminator, PRINT, 204  
THEN, 244,275,296  
TIMES\$, 266,298  
TOF, 159  
tokenization, 42  
tokenized, 45  
tone generation, 253  
Too Many Files, 324  
trigonometric function,  
    91,249,265  
TROFF, 267,296,308  
TRON, 57,58,267,297,308  
turtle graphics, 108,292  
TYPE, 268,301  
type code, 268,273  
type declaration, 27  
Type mismatch, 210,324

**U**

Undefined line number, 240,324  
Undefined user function, 324  
Unprintable error, 324  
UNTIL, 224,283,297,308  
UP, 269,297,308  
user subroutine, defining, 104

USING,  
    166,205,213,236,270,298,30  
    7,308  
USR, 70,104,271,280,297,301  
USR11, 274,277

**V**

VAL, 210,275,279,301,307  
variable name, 2,27,28  
variable storage format, 302  
variable type, 3  
variable type, declaring, 101  
variables, 47  
variables, access from  
    assembler, 312  
variables, clearing, 76  
variables, listing active, 33  
variables, passing, 73,87,239  
variables, swapping, 261  
VARPTR, 70,277,280,301,303  
video screen scroll,  
    107,151,232,269  
video screen zones, 264  
video screen, clearing, 79

**W**

WAIT, 282,297  
warranty, 11  
WEND, 283,297  
WEND without WHILE, 324  
WHILE, 283,297  
WHILE without WEND, 324  
WIDTH, 284,297  
WINKEY\$,  
    109,146,195,285,298,307  
WPEEK, 280,286,301,308  
WPOKE, 200,287,297,310  
WRITE, 288,297  
write, 45

**X**

XFIELD, 121,191,218,297,308  
XOR, 275,282,302,306,313

**Y**

YS, 58  
YX, 58,267

**Z**

Z180, 139,193,282  
Z80, 58,273  
Z80 assembler, 12,311  
Z80 Assembler mode, 13  
Z80 assembly language, 128  
Z80 source code, 12  
Z80-MODE,  
    13,54,63,86,128,149,247,25  
    3,311



