ZAPPLE™ 12K BASIC

USER'S MANUAL

PM-G12014-0300-01

TDL BASIC VERSION 3

USER'S MANUAL

(Manual Revision 0)

First Printing January 1978

PREFACE

This manual describes the Basic programming language, occupying slightly more than 12k of core, as implemented on a Z-80 based microcomputer.

It describes the unique and powerful commands available in TDL Basic Version 3. Also discussed are the tremendously powerful I/O handling capability, relocatability, ROMability and the large measure of hardware independence.

Great care has been taken to eliminate errors and omissions in this manual. Our technical support staff is available regarding any problems or questions you may encounter. Any effects however, or damages (including consequential) caused by reliance on the material presented, including but not limited to typographical, arithmetic, or listing errors, shall not be the responsibility of T.D.L.

Table of Contents

0.0 INTRODUCTION

0.1 What is a microcomputer?

Microcomputer is a term used to describe any computer built around a microprocessor. Recent advances in integrated circuitry have made it possible to put highly complex functions in a package the size of a domino. The idea of the microprocessor was to make what amounted to a CPU on a single chip. This chip, used as a universal process controller, would be replacing large amounts of complex circuitry in all sorts of equipment.

0.2 What is Basic?

Basic is a programming language most often supplied with small computers. It is very easy to learn, yet offers a great deal of programming flexability, having been originally designed as a beginner's language.

CHAPTER 1

GENERAL INFORMATION

Basic V3 functions as a BASIC Interpreter occupying slightly more than 12k of memory, and provides some of the most advanced software features of any commercially available Basic. If you have already worked with the Zapple 8-K Basic, you will appreciate the new commands that have been added and the extended capabilities of the existing commands.

Basic V3 offers many unique features, including user programmable error handling routines which can process any error occurring in the Basic program without aborting the program; serial input and output of ASCII or binary data files from the Zapple Monitor defined reader and punch devices, and the passing of a variable's address to an assembly language routine which allows routines to return data to the calling program.

All of the above are covered in this manual. This is NOT a "How to write Basic Programs" manual. Many excellent

texts on this subject have been produced. Your local Computer Store can recommend many such texts.

1.0 I/O HANDLING

Oftentimes the lament of the programmer is the lack of source documentation for a Basic Interpreter. This is usually due to the fact that internal I/O routines must be modified to suit the exact configuration of your hardware. The TDL method of I/O handling eliminates this problem in that the source code for the Monitor is provided, and once modified to your hardware configuration, ALL other TDL software automatically interfaces to your system.

Basic V3 has this feature of hardware independence. All of its I/O drivers are contained in the Monitor, so interfacing it to your hardware is simple.

To get the most out of BASIC, we highly recommend getting the 2K ZAPPLE MONITOR.

1.1 LOADING BASIC WITH ZAPPLE

Loading of Basic V3 is very straight-forward. It is loaded using the "R" command of either the Zap or Zapple Monitors. It is provided on paper tape in TDL's relocatable hex file format. It occupies slightly more than 12K of core.

Basic V3 has been assembled on TDL's relocating macro-assembler. Because of this, Zapple Basic is completely relocatable. It is not necessary to load and run this program at one address only. Within limits, which will be mentioned here, it may be loaded and run at any convenient address by the user.

The procedure for loading the program is very simple. Place the tape in the reader device on the nulls between the serial number and the start of the data. Type on the console: "R,(x)"(cr): and start the reader.

Example: R,300 (cr)

will load Basic V3 at address 300H. For the exact details on the operation of the "R" command, see either the Zap or Zapple Monitor Manuals.

After loading, Basic V3 will NOT sign on. You must begin execution at the address given in the relocation parameter above by typing: G300 (cr). Basic will then ask: "Highest Memory?" asking the user to type in decimal the upper limit Basic will be allowed to use. A carriage return (cr) will assign all available memory to Basic save for a small amount at the top which is reserved for use by the monitor.

The limits on its practical relocatability are governed by two factors; The buffer storage area required, and the address at which the monitor will be located.

The first factor is that, Basic V3 requires a buffer space of approximately 512 bytes. Regardless of the loading address of Basic, this 512 byte buffer resides from address 100H to 2FFH. Thus, the minimum loading address for Basic V3 is 300H. From this it should be evident that this Basic at no time uses any memory below address 100H, as is the case of 8-K Basic.

THE MINIMUM LOADING ADDRESS FOR BASIC V3 IS 300H.

As to the second factor, the Monitors are also relocatable, but we do recommend that they be placed up near the top of memory at F000 (up "out of the way"). Thus the 2K monitor would reside from F000 to F7FF (hex) allowing F800 to FFFF for monitor extension routines. (Such as a VDM driver, Tarbell driver. Etc.). Thus, since the Basic V3 occupies slightly more than 12K of core, the maximum practical loading address is B000 (hex).

Basic will normally use the location of the initialization routine as an input buffer, thus once Basic signs on, and you type in a program you must reenter Basic at the recovery point (loading address +3). If, during the sign on sequence, Basic gets the highest memory address that is below the beginning of Basic then Basic assumes that it is running in ROM and does not attempt to modify itself.

1.2 ZAPPLE BASIC VERSION 3 JUMP TABLE

All I/O handling for Zapple Basic is done through either the ZAP (1K) or ZAPPLE (2K) monitors. The I/O interfacing is done in the beginning of the Basic V3 Program.

Zapple Basic has in addition, a recovery address, from which recovery of the program can often be made following a

"blow-up".

Zapple Basic V3 also has a "USR" command which allows one's own assembly language routines to be called as part of a Basic program. A jump vector is provided allowing the user great latitude in this application.

On Page 5, the source code of the first part of Zapple Basic V3 is presented. It contains all of the I/O vectors which are necessary for complete user versatility. Note that as part of the code, addresses marked with an apostrophe (') are those addresses which are relocatable. Those without an apostrophe (') are considered absolute, in that they are vectoring to addresses outside of Basic, where they expect to find specific I/O routines.

The specific I/O routines in question are those of the monitor. Although both the Monitor and Zapple Basic are relocatable, we recommend placing the monitor (either the Zap or Zapple) as high as possible - usually F000H. Thus Basic expects to find the monitor at that address. The source code of these "jumps to the monitor" are presented so that in the event that you do not wish to, or are not able to have the monitor reside at this address, you may make the necessary, although simple modifications to the program.

Note that the source code below is in TDL's relocating assembler format, in that address information is presented in the "High byte first, low byte second" format. For example, at address 000C' there is a jump to F006 (hex). With some assemblers this might be construed to be a jump to address 06F0. Also note that any modifications you might make when using the monitor at an address other than F000 would entail changing only the high byte of the stated jump address.

```
0000' C3 XXXX'    BASIC:   JMP   INIT    ;"INITIALIZE"
                                           ENTRY POINT
0003' C3 XXXX'    REST:    JMP   RECOVER;RECOVERY ENTRY
                                           POINT
0006' C3 XXXX'    USR:     JMP   ERROR   ;USER DEF.
0009' C3 F003     CI:      JMP   CIN     ;CONSOLE INPUT
000C' C3 F006     RI:      JMP   RIV     ;READER INPUT
000F' C3 F009     CO:      JMP   CON     ;CONSOLE OUTPUT
0012' C3 F00C     PO:      JMP   WRTV    ;PUNCH OUTPUT
0015' C3 F00F     LO:      JMP   LISTX   ;LIST OUTPUT
0018' C3 F012     CSTS:    JMP   CSTSX   ;CONSOLE
                                           STATUS CHECK
001B' C3 F015     IOCHK:   JMP   IOCHX   ;I/O CONFIG.
                                           CHECK
001E' C3 F018     IOSET:   JMP   IOSTX   ;I/O MODIFCTN.
0021' C3 F01B     MEMSIZ:  JMP   MEMCK   ;MEMORY SIZE CK
0024' C3 F01E     TRAP:    JMP   TRAPX   ;BREAKPOINT ENTRY
```

"X"s are inserted into some jump notations above because the values may change in future versions of Basic, and thus could cause confusion. These addresses are modified in the course of various applications, and all that is needed is the recognition that they lie at the starting address of basic (where it was loaded), plus the address at the beginning of the line.

Specifics on making use of the USR command portion of the above are covered in the section of the manual which deals with the USR command.

1.3 ERROR MESSAGES

The following is a list of error messages returned by Basic when the particular error has been detected. They are given here without explanation. Within the context of a Basic program they indicate clearly what is amiss, and are of great use in program debugging.

```
 1   NEXT W/O FOR
 2   SYNTAX ERROR
 3   RETURN W/O GOSUB
 4   OUT OF DATA
 5   ILLEGAL FUNCTION
 6   ARITHMETIC OVERFLOW
 7   OUT OF MEMORY
 8   UNDEFINED STATEMENT
 9   SUBSCRIPT OUT OF RANGE
10   RE-DIMENSIONED ARRAY
11   CAN'T /0
12   ILLEGAL DIRECT
13   TYPE MIS-MATCH
14   NO STRING SPACE
15   STRING TOO LONG
16   TOO COMPLEX
17   CAN'T CONTINUE
18   UNDEFINED USER CALL
19   FILE NOT FOUND
20   ILLEGAL EOF
21   FILES DIFFERENT
22   RECOVERED
23   FNRETURN WITHOUT FUNCTION CALL
24   MISSING STATEMENT NUMBER
25   RECORD TOO LARGE
26   UNDEFINED MATRIX
27   INVALID UNIT NUMBER
28   RESUME W/O ERROR
```

CHAPTER 2

2.0 BASIC VERSION 3 COMMAND SET

COMMAND	GROUP	PURPOSE
ABS	8	Absolute Value Function
ALOAD	11	Loading of ASCII Source Programs
ALOAD*	11	" " " "
AMERGE	11	" " " "
AMERGE*	11	" " " "
AND	6	Logical AND operator
ASAVE	11	Allows punching of ASCII text.
ASC	9	Convert character to numeric value function
ATN	7	Arctangent function
AUTO	1	Provides automatic generation of line numbers while a Basic program is being entered.
CALL	6	Invokes assembly language subroutines
CHR$	9	Convert numeric value to character function
CLEAR	1	Delete all variables and set string space
CONT	1	Continue program execution from program breakpoint
COPY	5	Provides method of moving, or duplicating one section of a Basic program into another part of the program
COS	7	Cosine function
DATA	5	Defines constants
DEF	8	Defines User functions
DELETE	1	Deletes range of line numbers
DIM	5	Reserves storage for matrices
EDIT	2	Invokes the line editor
ELSE	6	What to do if relational is not true
END	10	End of program; return to command mode
EOF	6	Causes a software controlled initiation of the end of file processing.
ERL	14	Function to return the line number at which an error occurs.
ERR	14	Function to return the line number of error which last occurred.
ERROR	14	Allows the use of software generated errors in conjunction with the error trapping capability.
EXCHANGE	5	Exchanges the values of two variables without the use of a third variable.

COMMAND	GROUP	PURPOSE
EXP	8	Function to return "e" raised to a power
FN	8	Class of user defined functions
FNEND	8	Multiline functions
FNFAC	8	" "
FNRETURN	8	" "
FOR	6	Sets up a loop
FRE	8	Function to determine amount of unused memory.
GOSUB	6	Invokes a subroutine
GOTO	6	Transfer control to another part of the program
IF	6	Relational test
INP	5	Input directly from an I/O port
INPUT	5	Input data from the keyboard
INSTR	9	Searches one string for a specified substring
INT	8	Function returns the integer portion of a number
KILL	1	Allows unneeded matrix space to be returned to the system
LEFT$	9	Returns the left portion of a string
LEN	9	Returns the length of a string
LET	5	Logical Assignment
LINE INPUT	5	Provides increased flexibility in handling console input.
LIST	3	Lists the program on the console
LLIST	4	Lists the program on the list device
LLVAR	4	Lists the program variables on the list device
LNULL	4	Sets the nulls for the printer
LOAD	1	Loads a program from the reader
LOADGO	1	Allows one Basic program to load and transfer control to another.
LOG	8	Returns the natural logarithm of a number
LPOS	4	Function to return the current position of the list device
LPRINT	4	Outputs on the list device
LVAR	3	Prints the variables on the console
LWIDTH	4	Sets the width of the list device
MID$	9	Returns the middle of a string
MLOAD	5	High speed input of an entire numeric matrix at one time, in binary.
MSAVE	5	High speed output of an entire numeric matrix at one time, in binary.
NEW	1	Clears all program statements and variables
NEXT	6	Returns to the beginning of a loop
NOT	6	Logical "NOT" operator

COMMAND	GROUP	PURPOSE
NULL	3	Sets the nulls for the console
ON	6	Indexed transfer of control
ON EOF GOTO	6	Allows the user to detect and process the end of file condition.
ON ERROR GOTO	14	Specifies a user error handling procedure.
OR	6	Logical "OR" operator
OUT	5	Output directly to an I/O port
PEEK	5	Function to return data from a memory location
POKE	5	Insert data into a memory location
POS	3	Function to return the correct print head position of the console
PRECISION	1	Allows the specification of a default print-out precision of less than 11 digits
PRINT	3	Directs output to console
PRINT USING	3	String and numeric specifications
QUOTE	5	Provides output which is directly readable by an INPUT statement.
RANDOMIZE	8	Changes the seed used by the pseudo-random number generator
READ	5	Move data from a DATA statement to a variable
REM	10	Remarks
RENUMBER	1	Renumber the program and change line number references
RESTORE	5	Returns pointer to the beginning of the data statements
RESUME	14	Routine to return control to regelar program execution after error processing.
RESUME NEXT	14	Starts execution at statement following the one causing the error.
RETURN	6	Return control back from a subroutine
RIGHT$	9	Function to return the right portion of a string
RND	8	Function to return a pseudo-random number
RUN	1	Clear variables and start execution of program
SAVE	1	Dump a copy of the program to the currently assigned punch device
SGN	8	Function to return the sign of a variable
SIN	7	Sine function

COMMAND	GROUP	PURPOSE
SPC	3	Used in PRINT statement to print spaces
SQR	8	Function to return the square root of a number
STEP	6	Used in FOR statement for increment of loop control
STOP	10	Used to terminate program execution
STR$	9	Function to convert a value to a character string
SWITCH	3	Used to change the console assignment
TAB	3	Used in a PRINT statement to tab to a position
TAN	7	Tangent function
THEN	6	What to do if relational IF is true
TO	6	Used in FOR statement to specify limit
TRACE	3	Used to turn ON/OFF line number trace
USR	10	May be patched to user provided routine
VAL	9	Function to return the numeric value of a string expression
VARADR	6	Function to allow the actual address that a particular variable resides at in memory be obtained.
WAIT	5	Used to loop on a status port
WIDTH	3	Set the width of the console
WRITE	5	Binary output statement to write to output device.
HEXADECIMAL CONSTANT	9	Constant used directly in the Basic program
?	13	Same as PRINT
UP-ARROW	13	Exponentiation operator
-	13	Subtraction operator
*	13	Multiplication operator
/	13	Division operator
+	13	Addition operator
<	13	Less than operator
>	13	Greater than operator
=	13	Equals operator

COMMAND	GROUP	PURPOSE
CONTROL U	12	Delete input line
CONTROL C	12	Abort execution of program
CONTROL X	12	Return to monitor
CONTROL O	12	Suppress console output
CONTROL R	12	Allows more input to be entered
CONTROL T	12	Prints line number of line currently being executed
CONTROL S	12	Temporarily stop execution
CONTROL Q	12	Restart the program
RUBOUT	12	Delete previous character
, (comma)	12	Move to next TAB position or delimiter
;	12	Don't move
:	12	Used for multiple statements per line

NOTE: The control characters and operators listed at the end of the COMMAND SET LIST, while not exactly being BASIC commands, are included here for your reference and convenience.

CHAPTER 3

3.0 DETAILED DESCRIPTIONS OF COMMANDS/FUNCTIONS

All numeric calculations are carried out to twelve significant digits, and rounded to eleven digits. This includes all intrinsic functions (eg. TAN, SIN, etc.).

Each statement line may be up to 255 characters long. The line may be formatted for additional readability with both tab and space characters, and may be spread over multiple physical lines to emphasize program structure by use of the line-feed character. A tab character will cause the line, when listed, to be spaced to the next multiple of eight column. A line feed will list as a carriage-return/line-feed. Tabs, extra spaces, and line feeds are all ignored during the processing of the statements.

For example:

```
100<tab>IF I=23 THEN<line feed><tab>I=0:<line feed><tab>J=1
```

will list as:

```
100     IF I=23 THEN
        I=0:
        J=1
```

The statements would be executed as if they had been entered all on the same line.

An error does not cause a loss of program context. Even if the error causes a program abort, all currently active function calls, FOR loops, GOSUBS and error traps are preserved. It is then possible to examine or modify variable values or examine program statements, and to restart the program through a direct mode GOTO command. Any modification to the program itself will result in the loss of all variables and the program context. (See GROUP 14 ERROR HANDLING).

File number is used where <file number> is 0=console, 1=punch, and 2=list device. I/O list is used where <I/O list> is a list of variables specifying where to get the data from/or where to put the data.

3.1 GROUP 1 GENERAL PURPOSE UTILITY COMMANDS

AUTO

The AUTO command provides for automatic generation of line numbers while a BASIC program is being entered. The format of the command is:

AUTO [<starting number>] [,<increment>]

If <starting number>] is omitted, 10 is assumed, and if <increment> is omitted, 10 is assumed. To terminate the automatic numbering, an empty line (just a carriage return) should be entered. If a line is entered whose generated number is the same as an existing line, the new line replaces the old one.

CLEAR

Deletes all variables in storage. The command CLEAR followed by an argument, such as, "CLEAR 400" will set the string space to that value.

The CLEAR command may be placed in a program,

For example:

15 CLEAR 250

to set the string space to the exact amount needed by that program. If the argument is omitted, the string space is not changed.

CONTINUE

If your program is stopped by typing a control C or by executing a stop statement, then you may resume execution of your program by typing CONTINUE OR CONT. Between the stopping and restarting of the program you may display the value of the variables, (See PRINT and LVAR) or change the value of the variables, (see LET). However, you may not modify the program, or continue after an error.

DELETE

Deletes a range of line numbers. The DELETE command is followed by two line numbers separated by a dash "-". For example, DELETE 115-135 would delete from your program the line numbers 115 up to and including 135.

DELETE 25 would delete only line 25.

KILL

The KILL command allows unneeded matrix space to be returned to the system. The format of the command is:

KILL <matrix>1 [,...,<matrix n>]

Each name specified must be a matrix id with no subscripts following. Reference to a matrix which has not been defined is not an error. If the matrix has been defined, all the space being used by the matrix for variable storage and for matrix information will be returned to the system, and the matrix will be undefined.

LOAD

Loads a program from the reader device. The LOAD command is followed by a string expression which evaluates to a single character program id. A NEW operation is performed (see NEW), then the reader device is searched for a program under that name, if found, the program is then loaded. Example: LOAD "P"

A "bell" will sound on the console when the file starts to load. Additionally, a saved file may be verified by reloading the file using the following format:

LOAD?"P"

If an error occurs a message will be generated, otherwise, you will return to the command mode.

Example: A$="X"

LOAD A$
or
LOAD "X"

LOADGO

The LOADGO command is used to allow one Basic program to load and transfer control to

another. The format of the command is:

LOADGO <file id>[,<starting line>]

The command functions similarly to the LOAD command. When executed, it searches the reader device for an internal format program whose id is a string expression which evaluates to a single character program id <file id>. If found, the program is loaded, and control is transferred to it. The new program either begins at its first statement, or if the optional argument was given to the LOADGO command, at the specified line number.

Example:

LOADGO "P",1000

It is important to note that the LOADGO command clears the program area before initiating the file search. Also, all data values are cleared prior to the loading of the new program.

NEW

This command deletes all program statements and any stored variables. The slate is wiped clean, so to speak.

PRECISION

Because the added precision of this version of BASIC sometimes can produce long fractional results, the PRECISION command allows the specification of a default print-out precision of less than 11 digits. The format of the command is:

PRECISION [<digits>]

If <digits> is zero or omitted, the normal precision of 11 digits is restored. Otherwise, the precision is set as specified. This precision affects all numeric output not specified by a PRINT USING format. The internal calculations are still performed to the same accuracy (11 digits). The number is rounded to the desired precision before display.

RENUMBER

The RENUMBER command causes the lines of a program to be renumbered and all the internal line number references such as 123 GOTO 547 to be properly adjusted. This command has three parameters separated by a comma. The first parameter specifies the starting point for the line numbers, the second parameter specifies the increment between numbers. If either parameter is omitted it defaults to 10. The third optional argument specifies the current line number of the first line to be renumbered. The use of this option allows a "hole" to be inserted into a program to allow for the insertion of additional lines.

Format:

RENUMBER [<new number>][,<increment>]
[,<start line>]

Example:
```
RENUMBER
RENUMBER 10
RENUMBER ,10
RENUMBER 10,10
```

All start at 10 and increment by 10.

```
RENUMBER 40,10,60
```

Starting at line 60, change line 60 to 40 and renumber by 10.

RUN

RUN clears all variables and starts the execution of the program starting with the first program statement. RUN followed by a line number will clear all variables and start execution at that line number.

Example:
```
RUN 105
```

SAVE

This command causes a copy of the program to be output to the punch device using Basic's internal compressed format. The SAVE command has one parameter, a string expression which evaluates to a single character program id. The program name is used in reloading the program with the LOAD command. Also the comment string in the SAVE command may now be an arbitrary string expression, rather than

just a string constant.

Example: SAVE "P","<message>"

saves the program under the name "P".

Note: Only the 26 uppercase characters are valid as program names. All other characters will generate a SYNTAX ERROR message.

3.2 GROUP 2 THE EDIT COMMAND

EDIT The EDIT command followed by a line number invokes the line editor to process that line. The editor mofes a copy of the line to be edited into its edit buffer. At the end of the editing process, the user has the option of replacing the line in the program with the contents of the edit buffer, or throwing away the changes (say that you decide that you really don't want to make those changes).

Example: EDIT 55

The editor will then print the line number and then wait for single letter commands. ALL commands are NOT ECHOED. Illegal commands will echo as a bell. Some commands may be preceeded by a numerical instruction to repeat itself. These are shown by a lower case "n" and may range from 1 to 255.

EDIT COMMAND/ FUNCTION

A Reload the Edit Buffer from the program line. This is used after making a mistake.

nD DELETE "n" characters

E END EDIT - don't print line and replace program line with the contents of the edit buffer.

nFx FIND the "n"th character X in the edit buffer and stop with the pointer just before the character.

H DELETE everything to the right of the pointer and go to the insert mode.

I INSERT all following characters from the keyboard, STOP INSERTING on a carriage-return or Escape.

nKx KILL or DELETE characters from where the pointer is now to the "n"th character X, but don't delete that character.

L Print the line and return to the beginning of the line.

Q	QUIT. Leave the edit mode - without replacing the program line.
nR	REPLACE the "n" following characters with characters from the keyboard.
X	MOVE the pointer to the end of the line, and go to the insert mode.
SPACE	MOVE the pointer to the right.
RUBOUT	MOVE the pointer to the left.
CARRIAGE RETURN	End Editing, print the line, replace the program line. This may also be used to terminate the insert mode.
ESCAPE	END insert mode or cancel pending commands.

In the following edit examples an exclamation mark (!) is used to show the position of the console print head or cursor. This line:

```
55 PRINT A,B;"DOLLARS"
```

will be used in all the examples.

```
USER TYPES: MACHINE RESPONDS:

EDIT 55      55!

TO LIST OUT THE LINE: (command not echoed!

L            55 PRINT A,B;"DOLLARS"
             55!


TO MOVE THE POINTER FORWARD:

             (space)  55 P!
             (space)  55 PR!
             10(space)55 PRINT A,B;"D!
```

TO MOVE THE POINTER BACKWARD, THE SYSTEM ECHOS CHARACTERS THAT ARE PASSED BY THE POINTER AS IT IS GOING BACKWARDS.

```
(rubout)   55 PRINT A,B;"DD!
2(rubout)  55 PRINT A,B;"DD";!
L          55 PRINT A,B;"DD";;"DOLLARS"
           55 !
```

Although the example of what happens when using the rubout command may be confusing when shown as above, in actual use, the response of the machine when you type the rubout command is quite easy to get used to.

TO DELETE A CHARACTER:

```
D          55 \P\!

L          55 \P\RINT A,B;"DOLLARS"
           55 !

L          55 RINT A,B;"DOLLARS"
           55 !

(space)    55 R!

2D         55 R\IN\T A,B;"DOLLARS"

L          55 R\IN\T A,B;"DOLLARS"
           55 !

L          55 RT A,B;"DOLLARS"
           55 !
```

TO RECOVER FROM A MISTAKE:

At this point we decide we didn't really want to change the word "PRINT", so we can reload the edit buffer with the "A" command.

```
A          55 !

L          55 PRINT A,B;"DOLLARS"
           55 !
```

The INSERT command inserts characters into the line at the present position of the pointer. The INSERT command is terminated by either an escape character or a carriage return.

```
L             55 PRINT A,B;"DOLLARS"
              55 !

18(spaces)    55 PRINT A,B;"DOLLARS!

IXX(escape)   55 PRINT A,B;"DOLLARSXX!

L             55 PRINT A,B;"DOLLARSXX"
              55 !

L             55 PRINT A,B;"DOLLARSXX"
              55 !
```

The "X" command moves the pointer to the end of the line and goes into the insert mode.

```
XYY(escape)   55 PRINT A,B;"DOLLARSXX"YY!

L             55 PRINT A,B;"DOLLARSXX"YY
              55 !

L             55 PRINT A,B;"DOLLARSXX"YY
              55 !
```

The "H" command deletes all the line to the right of the pointer and goes to the insert mode.

```
11(spaces)    55 PRINT A,B;"!

HCENTS"(esc)  55 PRINT A,B;"CENTS"!

L             55 !

L             55 PRINT A,B;"CENTS"
              55 !
```

3.3 GROUP 3 COMMANDS INVOLVING THE CONSOLE

LIST

Causes the program to be typed on the console device. As a logical extension of the file number concept, the output control command LIST will accept an optional file number. List may have one or two parameters separated by a dash.

Format:

LIST #<file number>,[line number-line number]

An omitted file number will default to the console (0).

Example: LIST #2,20-30

will print lines 20 thru 30 on the list device. LIST 20 would only print line 20. LIST 20- lists from 20 to End.

LVAR

Causes the variable storage area to be typed on the console. As a logical extension of the file number concept, the output control command LVAR will accept an optional file number as follows:

LVAR #<file number>

An omitted file number will default to the console (0).

NULL

Sets the number of nulls to output to the console after a carriage return line feed sequence. This may be used to give additional time after a carriage return for terminals which require additional time to return the print head.

The NULL command may be followed by upto 3 parameters separated by a comma. The first parameter is an optional file number. The second specifies the number of nulls to send after the carriage-return/line-feed sequence and the third, as a decimal number, specifies

what character is to be used. (initializes to zero or ASCII null).

Format:

NULL [#<file number>,] <number> [,<character>]

An omitted file number will default to the console (0).

Example: NULL 3,255

sends 3 rubout characters after a carriage-return/line-feed to be output to the console.

POS

This function is used to return the present position of the print head or cursor of the console device. The first position is considered to be zero (0). The function POS accepts an optional file number.

Format:

POS(<file number>)

An omitted file number will default to the console (0).

Example: 40 A=POS(B)

where B=0 for the console or 1 for the punch or 2 for the list device. The example above would take the present position of the print head as a number from 0 to 131 and place it in variable A.

NOTE: LPOS will take a dummy argument since it already implies the line printer as the list device.

PRINT

The PRINT command is used to direct printed output to the console device. An optional <file number> parameter can be used to direct output to any of the three devices (0=console, 1=punch, 2=list device). The default is the console. The PRINT command is followed by a list of variables,constants, or literals to be printed, separated by commas or semicolons.

These are the variables in the <I/O list>.

Output in ASCII mode uses the PRINT and PRINT USING commands in the following format:

PRINT [#<file number>,] [USING <format>,] [#<file number>,] [<I/O list>]

where <file number> is again an expression evaluating to a valid file number. It can be located at either point indicated when the USING option is used. Valid file numbers for ASCII output are 0 for the console, 1 for the punch, and 2 for the list device (Note that file 2 overlaps the Lxxxx commands. The Lxxxx commands will be removed in a future version of Basic, so the new format is to be perferred). <I/O list> is a list of variables specifying where to get the data from. The PRINT command outputs exactly the same format to any of the three devices.

Example: 40 PRINT #2,123,A,"IS THE ANSWER"

If the separators are commas, then the items are printed in columns spaced every 14 positions across the list device.

If the separators are semicolons, the items are printed with 2 spaces in between. Additionally, if the last item in the print list is followed by a semicolon, then unless executing the command would overprint the last print position on the console, BASIC will not carriage return but would stack successive PRINT commands across the console on the same line.

```
Example:  10 PRINT A,B;
          20 PRINT C
```

would print A,B and C across the same line on the console. There would be 14 spaces between the beginnings of A and B and 2 spaces between the end of B and the beginning of C.

PRINT USING The PRINT USING statement has two formats available:

PRINT USING [#<file number>,] <line number>[;<I/O list>]

PRINT USING [#<file number>,] <string value>[;<I/O list>]

In the first format, the <line number> refers to a "format line" which contains the format string. This line must start with an exclamation point (!) and be followed by the format specification. In the second format, the format specification is taken from the string value specified.

For example:

```
100  A = 5
110  PRINT USING 120;A
120  !##.##
130  PRINT USING "##.##";A
```

Both lines 110 and 130 would print a space followed by the characters 5.00

The format specification consists of any valid combination of the following field specifiers:

FORMAT SPECIFICATIONS

NUMERIC

\# Numeric fields are specified by use of the #. Each # in a field represents one digit position. The number will be right justified in the field, with leading spaces added to fill the field.

. Decimal point alignment is specified by the use of a decimal point. The number will be rounded to fit the specification. A digit before the decimal point will always be filled (with a zero if necessary). For example:

```
###.## 23.456 =>  23.46
###.### -24.5 => -24.500
###.## .12345 =>   0.12
```

\+ A plus sign may be used either at the start or the end of a numeric specification. It will force the + sign to be printed at that end of the field if the value is positive (normally a space would be printed). A - sign will be printed in that position if the number is negative.

- A minus sign may be used at the end of the numeric specification to indicate that - sign for a negative number should be printed at the end, as opposed to the start, of the number. If the number is positive, a space is printed.

** Two (or more) asterisks at the start of a numeric field indicate asterisk "fill" of the field. Each asterisk indicates one digit position, and all empty digit positions prior to the decimal point will be filled with asterisks instead of spaces.

$$ Two (or more) dollar signs at the start of a numeric field indicate a "floating" dollar sign. This means that the dollar sign will be placed immediately adjacent to the first non-zero digit in the number. Each dollar sign indicates one digit position, but one of these positions is occupied by the dollar sign itself.

**$ This indicates the combination of the two above features.

, A comma anywhere to the left of the decimal point indicates that commas are to be inserted every three digits. Each comma also indicates one digit position in the specification.

^^^^ Four up-arrows at the end of a numeric specification indicate that the number is always to be printed in exponential notation. Decimal point alignment is allowed, but the significant digits are left justified and the exponent is printed accordingly (E+nn or E-nn).

If any numeric value will not fit in the field specified, it will still be printed in its entirety, but will be preceeded by a percent sign (%).

STRING

String fields are specified by use of the apostrophe ('). A single apostrophe indicates a single character string field. Multiple character string fields are specified by following the apostrophe with one (or more) of the letters C, R, L, or E. The size of the field is equal to one plus the number of letters following. The letters have the following meanings.

L The string value is left justified in the specified field. If the value is longer than the field, extra characters are lost on the right.

R The string value is right justified in the specified field. If the value is longer than the field, extra characters are lost on the left.

C The string value is centered in the specified field. If the value is longer than the field, extra characters are lost on the right.

E The string value is left justified in the specified field. If the value is longer than the field, the field is Extended to allow the entire string value to be printed with no lost characters.

Note that the format string will be reused until all values in the output list have been printed. The printing of the format string will terminate when a field specification is encountered and the output list is empty. Any character in the format specification which is not part of a numeric or a string specification will be printed literally on the output device at the specified position in the string.

SPC This is a function-like command that is used to print a number of spaces on the console. It is only used in a PRINT or LPRINT statement and is called function-like because it looks like a function but CANNOT be used in a LET statement.

Example: 35 PRINT A;SPC(5);B

may be used to place an additional 5 spaces between A and B over and above the two that would normally be printed due to the semi-colon.

SWITCH This command is used to change the console assignment. SWITCH used with no variable will always switch between the teleprinter and the user console. SWITCH with an argument of 0-3 (zero to three) will assign the console to that value. I.E.

0=TTY
1=CRT
2=BATCH MODE
3=USER DEFINED

A value greater than 3 will generate an error message. These are further discussed in the ZAPPLE Monitor Manual.

TAB

TAB is a function-like command that is used only with a PRINT or LPRINT statement and is used to tab directly to a particular position. If the printhead or cursor is on or after the specified TAB position then BASIC will ignore the TAB command.

Example: 25 PRINT A;TAB(25);B

TRACE

This is a one parameter command that turns on or off the TRACE function. If the TRACE is on, then Basic will print the line numbers of the statements executed enclosed in angle brackets, e.g. <25>. The parameter may be an expression and if that expression is evaluated to be non-zero, then the TRACE is turned on. If the expression is evaluated to be zero, then the TRACE is turned off. (NOTE: The TRACE and LTRACE are completely independent functions and may be separately manipulated at will.)

Example: 25 TRACE A-B

If A-B is equal to zero then the TRACE is turned off, if equal to non-zero then TRACE is turned on.

WIDTH

Basic keeps track of the number of characters and spaces printed on the console and will generate an automatic carriage-return/line-feed to prevent overprinting at the end of a line. The WIDTH command may be used to change the sign-on default of 72 spaces. WIDTH accepts an optional file number. (0=console, 1=punch, 2=list device)

Format:

WIDTH [#<file number>,] <width>

Example: WIDTH 80

will cause an automatic carriage-return/ line-feed sequence after 80 characters. The minimum value is 15, and the maximum value is 255.

3.4 GROUP 4 COMMANDS INVOLVING THE LINE PRINTER

Most of the commands of GROUP 3, which affect the console, have a counterpart in GROUP 4, which affects the line printer. The general rule is to add the letter L in front - thus PRINT becomes LPRINT, and LVAR becomes LLVAR, etc. This section simply lists the commands and directs your attention back to GROUP 3 for information on how the commands function otherwise. These commands will eventually be removed in a future version of Basic. The use of the <file number> which is an expression evaluating to a valid file number (i.e. 0 for the console, 1 for the punch and 2 for the list device) will take precedence over the Lxxxx commands.

LLIST	see LIST
LLVAR	see LVAR
LNULL	see NULL
LPRINT	see PRINT
LPRINT USING	see PRINT USING
LTRACE	see TRACE
LWIDTH	see WIDTH
LPOS	see POS
SPC	use in LPRINT statement
TAB	use in LPRINT statement

3.5 GROUP 5 COMMANDS AND FUNCTIONS THAT INVOLVE THE MOVEMENT OF DATA FROM ONE PLACE TO ANOTHER.

LET(=) This is the assignment command. It causes the evaluation of an expression on the right side of the equals sign (=) and the assignment of the resultant value to the variable on the left side of the equals sign.

Example: 10 LET A=B+2

would cause BASIC to get variable B, add 2 to it, and place the result in A. In TDL Basic, the command LET is optional. For example, the previous statement could also be written as:

10 A=B+2

DIM Reserves storage for matrices. The storage area is first assumed to be zero. Matrices may have from one to 255 dimensions, but is limited by the available remaining workspace (memory).

```
257  DIM A(72),B(4)
258  DIM C(72,66)
259  DIM D(J)
```

Matrices may also be dimensioned during the execution of the program after the storage space is calculated, however, remember that the DIM command zeros the storage area, and a previously dimensioned array may not be re-dimensioned.

DATA Specifies constants that may be retrieved by the READ statement.

Example: 10 DATA 5,4,3,2,1.5

READ Retrieves the constants that are specified in the DATA statement. Binary I/O is performed by the use of the READ and WRITE commands. Binary data is written in internal

format, and is only useful to other Basic programs. The format of the input command is:

READ #<file number> [,<I/O list>]

This command performs two functions, depending on whether an I/O list is present or not. If no I/O list is present, the command causes the input data stream to be searched until a binary data header is found (seven 0FFH followed by one 00H). The input is positioned at the first byte following the header, and the command is done. If an I/O list is present, then sequential bytes are read from the input device until the I/O list is satisfied. It is important to note that Basic does no checking on the validity of the incoming data. It is the users responsibility to read the data in a way compatible with that in which it was written.

Example: 20 READ #1,A

The first time line 20 is executed the value 5 from the previous example will be placed into variable A. If at some later time statement 20 is executed again, or another READ statement is executed, then the value 4 would be retrieved etc. DATA statements are considered to be chained together and appear to be one big data statement. If at any time all the data has been read and another READ statement is executed then the program is terminated and the message "OUT OF DATA @ LINE (N)" is printed.

RESTORE

This command restores the internal pointer back to the beginning of the data so that it may be read again. It takes an optional line number as an argument. If specified, the DATA read pointer will be set to the specified line instead of the start of the program. The format of the command is:

RESTORE [<line number>]

INPUT
LINE INPUT

INPUT allows the operator to type data into

one or more variables. The ASCII input command has the following format:

INPUT [#<file number>,] <I/O list>

where <file number> is the specification of where the input is coming from. Currently defined file numbers for input are 0 for the console, and 1 for the reader. The file number may be any arbitrary expression that evaluates to a valid file number. If the file number clause is omitted, then the console is assumed. The <I/O list> is a list of variables specifying where to put the data. ASCII input from either device must follow the normal rules for the specified input type. This means that input from the reader must be delimited by commas or carriage returns, and strings containing special characters must be surrounded by quotes.

Example: 35 INPUT A,B

would cause the printing of a question mark on the console as a prompt to the operator to input two numbers separated by a comma. If the operator doesn't type enough data then BASIC responds with 2 question marks.

```
Example:  10 INPUT A,B,C
          RUN
          ?5
          ?? 7,5
          READY
```

would input the value 5 to the variable "A" and when the operator typed carriage return, Basic wanted more data and so responded with 2 question marks.

The input statement may be written so that a descriptive prompt is printed to tell the user what to type.

```
Example:  10 INPUT "TYPE A,B,C";A,B,C
          RUN
          TYPE A,B,C? (ans)5,6,7
          READY
```

This causes the message placed between the quotes to be typed before the question mark. Note the semicolon must be placed after the

last quote.

The entry of just a carriage return as the response to an INPUT statement does not return it to command level. The empty line will correspond to a single value of zero if a numeric variable was being input, or the null string variable was being input. If more variables are left in the input list, additional input will be requested. To terminate program execution and return to command level, a control-C should be entered just as if the program was running.

The LINE INPUT statement provides increased flexibility in handling console input. The format of the statement is:

LINE INPUT [#<file number>,] <I/O list>

where <file number> is the specification of where the input is coming from. Currently defined file numbers for input are 0 for the console, and 1 for the reader. The file number may be any arbitrary expression that evaluates to a valid file number. If the file number clause is omitted, then the console is assumed. The <I/O list> is a list of variables specifying where to put the data. ASCII input from either device must follow the normal rules for the specified input type. This means that input from the reader must be delimited by commas or carriage returns, and strings containing special characters must be surrounded by quotes.

The statement functions similarly to the normal INPUT statement. The prompt string, if present, is output to the console. Each variable in the input list must be a string variable and is assigned the value of the entire input line as typed by the user, with no formatting. If more than one variable is present, then additional lines are requested. An empty line (just a carriage-return) has the value of the null string.

PRINT/LPRINT

PRINT and LPRINT are the converse of INPUT in that they print out data on the console and line printer. For an explanation of these

commands see GROUPS 3 and 4.

INP

Basic has the ability to directly read an input port. The INP function takes as its argument the number of the port to be read, and the result may be assigned to a variable or printed directly.

```
Example:  10 A=INP(0)
          20 PRINT INP(0)
```

would in both cases input from port zero. In line 10 the value input from the port is placed in variable "A" and in line 20 it is directly printed.

MLOAD

The input statement MLOAD is a high speed input mode for those users who have only non-controlled I/O devices. The format of the statement is:

```
MLOAD #<file number>, <matrix 1> [,<matrix 2>...]
```

The statement inputs an entire numeric matrix at one time, in binary. Each matrix is preceded by a binary data prompt. The speed of input is such that an uncontrolled device can be used. Each matrix must be defined prior to its use in the statement. Multi-dimensional arrays are stored and loaded in a sequence with the last subscript varying most rapidly. There is no requirement that a matrix be read back into the same size matrix it was written from, all correspondence is up to the user. A binary prompt is required before each array read however.

MSAVE

The output statement MSAVE is a high speed output mode for those users who have only non-controlled I/O devices. The format of the statement is:

```
MSAVE #<file number>,<matrix 1> [,<matrix 2>...]
```

This statement outputs an entire numeric (not string) matrix at one time, in binary. Each matrix is preceded by a data prompt. The speed of output is such that an uncontrolled

device may be used. Each matrix must be defined prior to its use in the statement. Multi-dimensional arrays are stored and loaded in a sequence with the last subscript varying most rapidly. There is no requirement that a matrix be read back into the same size matrix it was written from, all correspondence is up to the user. A binary prompt is required before each array read however.

OUT

This command causes Basic to output data directly to any output port. The OUT command has two parameters separated by a comma. The first parameter is the port number and the second parameter is the data to be output.

```
Example:  10 A=1
          20 B=7
          30 OUT A,B
          RUN
```

would cause a seven to be output to port one, and if your console data port is port one the bell would ring since a 7 is a BELL in ASCII code.

QUOTE

This statement is used for ASCII output devices. The format of this statement is:

QUOTE [#<file number>,] [<quote character>]

where <quote character> is either zero, omitted, or the decimal value of an ASCII character (as in the NULL command). If the value is zero (or omitted), then the output from the device will appear the same as in previous versions of Basic. If the value is non-zero, then the device will output in special QUOTE mode. In this mode, which only affects normal PRINT statements (not USING), all commas occurring in the I/O list of the PRINT statement will not cause the standard TAB function, but will be sent to the output device. In addition, any string variable printed will be preceeded and followed by the specified ASCII character (which is usually a double quote [decimal 34]). The effect of this mode is to provide output which is

directly readable by an INPUT statement.

Each of the three devices has defaults for each of the three characteristics. The console and list (0 and 2) devices have default widths of 72, null counts of 0, null characters of 0, and quote modes of 0. The punch device (1) has a default width of 253, null count of 0, null character of 0, and quote mode 34 (double quote).

WAIT

If you write a program to INP or OUT directly to the console for a purpose such as reading a paper tape from the teleprinter tape reader, Basic itself will interfere with inputting the data because Basic is looking at the console keyboard to see if a control-C is typed to abort execution or control-X is typed to return to the Monitor. The WAIT statement will place Basic in a loop, looking at a specified status port, until a specified condition occurs. Then and only then will the next statement be executed.

(NOTE: Be careful using this because it is possible to put Basic in a loop waiting for a condition that will never occur. Should this happen, your only recourse is to reset the machine, or examine the memory location 3 higher than the address the program was loaded at, and hit RUN again. Basic will then recover without destroying your program.)

Example:
```
100 WAIT A,B,C
110 D=INP(A+1)
```

Basic will then input port "A", EXCLUSIVE OR the value with "C", and then AND the result with B. If a zero result occurs, then the process is repeated until a non-zero result occurs. Basic, in this example, will input from the next higher port and place the data in "D". The fact that at Line 110 Basic looked at the console port to see if the data was a control-C would not affect the proper inputting of data. In this example the status port is 0, the data port is 1, the data available bit is bit 2, and it goes low (0) to indicate that data is available on port 1. Then let A=0 for port Zero and One. Let B=4

to isolate Bit 2, and let C=255 so that a complement of the status occurs to follow the rule that data available is indicated by a non-zero result. If parameter C is omitted, then Basic defaults to zero for the value.

WRITE

As in the READ statement, this statement performs two functions. If no I/O list is present, a binary data header is written on the punch device. If an I/O list is present, the data is written as specified on the output device. All strings include their length as part of the data written.

The binary output statement has the form:

WRITE #<file number> [,<I/O list>]

PEEK

This function allows the direct retrieval of data anywhere in memory.

Example: 50 B=PEEK(A)

causes the value of the byte at address "A" to be assigned to the variable "B". Address "A" may range from 0 to 65535 (decimal).

POKE

Has two parameters.

Example: 57 POKE A,B

in which the first parameter specifies an address in which to insert the data specified by the second parameter. The address may range from 0 to 65535 and the data may range from 0 to 255.

COPY

The COPY command provides a method of moving, or duplicating, one section of a Basic program into another part of the program. The format of the command is:

COPY<new line>[,<increment>]=<line range>

The command copies the set of lines specified by <line range>, which is in the same format as the LIST command to that part of the program specified by <new line>. The lines are renumbered as they are copied, starting with <new line> and incrementing by <increment> (which is 10 if omitted). Before copying any lines, the new line numbers are validated to guarantee that they will not overlap any existing lines, and that the new lines do not fall with the range of the lines being copied. Only the line numbers are changed, the lines themselves are not modified. The original lines are also not modified.

EXCHANGE

To speed sorting operations, the EXCHANGE command has been added. The format of the statement is:

EXCHANGE <variable 1>,<variable 2>

This statement exchanges the values of the two variables specified. Both variables must be predefined, and of the same type. A single matrix element may be used as either of the variables. For example:

```
EXCHANGE A$,B$
EXCHANGE A$(2,3),C$
EXCHANGE C,D(I,J)
```

This exchange of values is accomplished in the most rapid way possible (strings just switch pointers).

3.6 GROUP 6 TRANSFER OF CONTROL AND RELATIONAL TESTS

EOF

User initiated end of file processing. The EOF statement may be used by itself to cause a software controlled initiation of the end of file processing. The format of the statement is:

EOF

This allows an end of file to be determined based on some programmed condition as well as on a hardware detected one.

GOTO

This statement followed by a valid existing line number will cause Basic to transfer control directly to that statement.

Example: 55 GOTO 100

will, if line 100 exists, cause execution of the program to resume at line 100.

GOSUB

This statement acts in a manner similar to that of GOTO except that the location of the next statement is saved so that a RETURN can be performed to return control.

RETURN

This statement is used to "return" control back to the statement following the most previous GOSUB that control came from.

ON EOF GOTO

End of data file detection. If an end of file indication is obtained from the Zapple Monitor, or a ctl-Z (1AH) is read in ASCII mode, then the end of the data file has been reached. If no end of file action is specified, an error message will be given. To allow the user to detect and process the end of file condition, the ON EOF GOTO command is used. The format of the command is:

ON EOF GOTO [<line number>]

If the line number is omitted, then the EOF processing is disabled. After the execution of this statement, any end of file encountered on a data input statement (INPUT,READ, or MLOAD) will cause a GOTO to the line specified. Execution will then continue with the statement at that line. This end of file processing is not a subroutine type of call, in that there is no way to "return" to the input statement causing the condition. Also, the occurrence of an end of file branch does not disable the ON EOF, so a subsequent end of file will again cause the GOTO to be executed.

ON x GOTO
ON x GOSUB

The ON statement causes control to be transferred to the "x"th line number in the list.

Example: 10 ON A GOTO 100,125,150

If A was equal to 1, control would be transferred to Line 100. If A=2, GOTO Line 125, etc. If A is equal to zero, or larger than the number of line numbers in the list, control will be given to the statement after the ON x GOTO. The value of x may range from 0 to 255.

CALL

A greatly improved method of invoking assembly language subroutines is provided by the CALL statement. The format of the statement is:

CALL <address>[,<argument 1>[,...,<argument n>]]

The <address> is an expression representing the machine address of the routine to call. The statement also allows the optional specification of arguments to be passed to the subroutine. Each argument expression is evaluated and converted to a 16-bit integer. These arguments are then pushed onto the stack, so the number of arguments is limited only by memory. When the subroutine is entered, the following information is available to it:

```
SP ->
     <argument n>
     <argument n-1>
          ...
     <argument 1>

HL ->
     <return address>

BC ->   # of arguments on stack
```

The arguments may then be popped off the stack in reverse order. In addition, note that the HL registers point to the location of the return address in the stack. If no arguments are desired, or some error abort is required, a SPHL followed by a RET will properly clean up the stack and return to the calling program. The BC registers contain the count of the arguments passed to the routine.

FOR,TO,STEP,NEXT

These keywords are used to set-up and control loops.

```
Example:  10 FOR A=B TO C STEP D
          20 PRINT A
          30 NEXT A
```

If B=0, C=10, and D=2, the statement at line 20 will be executed 6 times. The values of A that will be printed will be 0,2,4,6,8,10. "A" represents the name of the index or loop counter. The value of "B" is the starting value for the index, the value of "D" is the value to be added to the index. If D is omitted then the value defaults to 1. The "NEXT" keyword causes the value of "D" to be added to the index and then the index is tested against the value of C, the limit. If the index is less than or equal to the limit, control will be transferred back to the statement after the "FOR" statement. The index may be omitted from the "NEXT" statement, and if omitted the "NEXT" statement affects the most recent "FOR". This may be of concern in the case of nested "FOR-NEXT" statements.

```
Example:  10 DIM A(3,3)
          20 FOR B=1 to 3
```

```
30 PRINT "PLEASE TYPE LINE";B
40 FOR C=1 to 3
50 INPUT A(B,C)
60 NEXT C
70 PRINT "THANK YOU.";
80 NEXT B
```

IF,THEN,ELSE

The "IF" keyword sets up a conditional test.

Example: 25 IF A=75 THEN 30 ELSE 40

Upon execution of line 25 if A is equal to 75 then control is transferred to line 30. Else if A is not equal to 75 transfer control to line 40. The "THEN" clause may be replaced by a GOTO.

Example: 25 IF A=75 GOTO 30 ELSE 40

The THEN and ELSE clauses may contain imperative statements.

Example: 30 IF A=75 THEN A=0 ELSE A=A+1

The ELSE clause may be omitted in which case control passes to the next statement.

Example: 40 IF A=75 THEN A=0

Relational operators used in IF statements:

```
=  EQUAL
<> NOT EQUAL
<  LESS THAN
>  GREATER THAN
<= LESS THAN OR EQUAL
>= GREATER THAN OR EQUAL
```

The logical operators may also be used:.

```
NOT Logical Negation
AND Logical And
OR  Logical Or
```

Example: 20 IF(A=0) OR NOT(B=4) THEN C=5

VARADR Variable address references. This function allows the actual address that a particular

variable resides at in memory to be obtained. The function format is:

VARADR (<variable>)

where <variable> is either a single variable or a matrix element reference. This function returns an integer value which is the address in memory at which the value of the variable or matrix element specified resides. This address may be used directly in the POKE and CALL statements, and in the PEEK function. The formats of the variable values are as follows:

Numbers: 6 bytes per number, least significant byte first. The first 5 bytes are the mantissa, with the sign in the 5th byte. The mantissa is stored in a sign/magnitude overnormalized form (the high order bit is always assumed to be 1, and is used to hold the sign bit). A sign bit of 1 is a negative value. The 6th byte contains the exponent, in excess 128 notation (the value is always positive = actual exponent + 128). It is a binary exponent. For example, the hex string 00 00 00 00 00 80 is the number .5.

Strings: 6 bytes per dope vector. The first byte contains the length of the string. The next byte is always zero. The next two bytes contain the address of the string itself, least significant byte first. The last two bytes are unused.

3.7 GROUP 7 TRIGONOMETRIC FUNCTIONS

ATN

Function to return the ARCTANGENT of a value. The result is expressed in radians.

Example: 10 B=ATN(.45)

returns the angle, expressed in radians, whose tangent is equal to .45.

COS

Function to return the COSINE of an angle, expressed in radians.

Example: 20 C=COS(A)

SIN

Function to return the SINE of an angle, expressed in radians.

Example: 30 D=SIN(A)

TAN

Function to return the TANGENT of an angle, expressed in radians.

Example: 40 T=TAN(A+B)

3.8 GROUP 8 MISCELLANEOUS FUNCTIONS

ABS

Function to return the absolute value of a value.

Example: 10 A=ABS(B+C)

If the result of the expression is positive then ABS returns that value. If the result is less than zero, then ABS returns the positive equivalent.

DEF,FN

These commands allow the user to define his own functions. A function defined in this way must have a name that begins with the letters "FN" followed by a valid variable name. For example "FNA", or "FNZ9". The function name is then followed by one parameter enclosed in parentheses. This parameter is a dummy argument and is included in the expression to the right of the equals sign.

Example: 159 DEF FNQ(X)=X*A

In this case A is a variable within the program and X is an argument that may be replaced with a constant or another variable when the function is used.

Example:
```
15 DEF FNQ(x)=X*A
 .
 .
121 A=3
122 B=4
123 C=FNQ(B)+5
```

Thus Basic would take the argument B and multiply it by the value of A, add 5 to the product and place the result in variable "C".

User defined functions may return both numeric and string values. Also, functions may have more than one (or less - zero) parameters, and the parameters may be either numeric or string. Secondly, functions may now consist of more than one statement (multi-line functions). The format of a multi-statement function is as follows:

```
DEF FN<function name>[(<dummy 1>[,<dummy n>])]
        ...
   <function body>
        ...
  FNEND[<function value>]
```

The first line of the definition is the same as the first part of a single line definition. The difference is the absence of the equal sign following the header information. The function definition header is followed by the actual statements comprising the function body. The function body may consist of any valid BASIC statements except another multi-line function definition (single line definitions are valid). The last statement of the function must be the FNEND statement. This statement both signals the end of the definition itself, and, upon execution, returns the value of the function to the calling expression. The FNEND statement takes a single optional argument which may be any valid BASIC expression of the same type as the function itself (string or numeric). The value of the expression is the returned value of the function. If no expression is given, the function returns a zero if numeric, or the null string if string.

To allow for the programatic termination of the function, and for the return of alternate values, the FNRETURN statement is provided. This statement has the form.

```
FNRETURN [<function value>]
```

The FNRETURN statement, when executed, functions identically to the FNEND statement. It terminates the function and returns the specified value. The difference is that a function may have multiple FNRETURN's, and they may occur wherever a valid BASIC statement may occur in the function.

The following are a few examples of multi-line functions:

CALCULATE A FACTORIAL

```
100 DEF FNFAC(I)
200 IF I=0 THEN FNRETURN 1 'FAC(0)=1
300 FNEND FNFAC(I-1)*I 'FAC(I)=FAC(I-1)*I
```

BUILD A REPETITIVE STRING

```
100 DEF FNREP$(I$,I)
200 J$=""
300 IF I<=0 THEN FNRETURN J$
400 FOR J=1 TO I
500 J$=J$+I$
600 NEXT
700 FNEND J$
```

It should be noted that a multi-line function may call any other function (including itself), may do GOSUB's to anywhere in the program, and may modify any program variable.

EXP

This function returns the base of the natural log system "e" or 2.71828 raised to a power.

Example: 20 B=EXP(A)

If A is equal to 1 the result is "e" or 2.71828.

FRE

This function, when used with a dummy variable, returns the amount of memory available for Basic Programs and variables, but which is currently unused. If FRE is used with a dummy string variable, it returns the amount of currently unused string space.

Examples: 30 A=FRE(X)
40 B=FRE(X$)

INT

Returns the integer portion of a number. This is essentially a "round-down" operation. For negative arguments the result would be the next more negative integer.

Example: 50 C=INT(D)

If "D" has a value of 5.25 then "C" will have a result of 5.0. If "D" has a value of -3.4, then "C" will be set to -4.0.

LOG

Returns the natural logarithm of the expression used as an argument.

Example: 60 E=LOG(F+G)

will return the log to the base "e" of the expression "F+G".

SGN

Will return the value +1 if the argument is greater than zero, zero if the argument is zero, and -1 if the argument is less than zero.

Example: 70 H=SGN(I)

SQR

Returns the square root of the argument& The argument may not be less than zero.

Example: 80 J=SQR(K)

RND

Returns a pseudo-random number in the range between 0 and 1. The RND function uses a dummy argument to perform the following functions. An argument less than zero is used to initialize the pseudo-random number sequence. An argument of zero will return the previous random number. An argument of more than zero will return the next pseudo-random number in the sequence.

Example: 90 R=RND(1)

RANDOMIZE

Although this is not a function, it is discussed here because of its relation to RND. The RANDOMIZE command may be used to generate a truly random starting point for the pseudo-random number sequence. (RND)

3.9 GROUP 9 STRING RELATED FUNCTIONS

ASC

Returns the decimal number that represents the first ASCII character of the string expression used as an argument.

Example: 10 A=ASC(A$)

The decimal value 65 represents an ASCII "A". If the character "A" was the left-most character of string "A$" then variable "A" would be set to 65.

CHR$

Returns the ASCII character represented by the decimal value of the argument.

Example: 20 PRINT CHR$(7)

would ring the bell on the teleprinter connected to the console. A "7" is an ASCII bell.

LEFT$

Uses two arguments, the first is the string expression and the second is the number of characters to return from the left end of that string. The second parameter may range from 1 to 255.

Example: 30 B$=LEFT$(A$,5)

would set B$ equal to the first 5 characters of string "A$".

The two string functions LEFT$ and RIGHT$ will allow a length argument of zero, resulting in the return of the null string as the function value.

LEN

Returns the length of a string expression in bytes.

Example: 40 X=LEN(S$)

would set "X" to the number of bytes contained in the string "S$".

MID$ May have 3 parameters:
1) The string expression.
2) The position to start extracting characters.
3) The number of characters to extract. This value defaults to 1 if omitted.

Example: 50 A$=MID$(B$,5,6)

would take 6 characters starting at the fifth character from the left of string "B$", and place that string in A$.

The MID$ function can appear on the left-hand side of an equal sign to specify the insertion of a sub-string into an already existing string variable. The format is:

MID$(<string variable>,starting char>[,<length>])=<string value>

The <string value> will overlay the characters in the value of the <string variable> starting at the specified character (the first character is number one) for the specified length. If the length is omitted, the entire remainder of the string is replaced. If the string value is smaller than the length specified, it is padded to the specified length with trailing blanks. If it is longer, the extra characters are ignored.

For example:

```
A$="123456789"
MID$(A$,3,5)="ABCDE" => A$="12ABCDE89"
MID$(A$,3,5)="AB"    => A$="12AB   89"
MID$(A$,3,5)="ABCDEF"=> A$="12ABCDE89"
```

This is significantly faster than any method using splitting and reconcatenation of the strings.

RIGHT$ See LEFT$ except works on the right-hand end of the string.

STR$ Returns a string whose characters represent the numeric value of the argument.

Example: 70 A$=STR$(2.2)

would return the characters "2.2" preceded by a space as the result.

VAL Opposite of STR$. Returns the numeric value represented by a character string.

Example: 80 A=VAL("4.5)

would return the numeric result 4.5.

HEXADECIMAL CONSTANT

Hexadecimal constants may be directly used in the BASIC program. Each constant must be preceeded by an ampersand (&). The constant must not exceed 65536 (&FFFF) in value. These constants may be used anywhere a regular numeric constant (not a line number) may be used (including as an argument to the VAL function).

INSTR This function searches one string for a specified substring. The format of the function call is:

...INSTR(<string value>,<string value>[,<start char>[,<length>]])...

In the basic function call, the first <string value> is searched to see if it contains t`e second <string value>. If so, the value of the function is the character position of the first character of the matched string. If no match is found, the value is zero. The two optional arguments correspond exactly to the application of the MID$ function to the first <string value> prior to the search. If a match is found however, the resulting value is still relative to the first character of the string. For example:

```
INSTR("123456789","456") => 4
INSTR("123456789","654") => 0
INSTR("1234512345","34") => 3
INSTR("1234512345","34",6) => 8
INSTR("1234512345","34",6,2) => 0
```

3.10 GROUP 10 MISCELLANEOUS COMMANDS

END — Stops the execution of the program. The END command may be placed anywhere in the program.

Example: 65520 END

REM — Denotes that this line is a remark and is not processed.

Example: 10 REM This is a remark.

' (REMARK) — Basic allows every statement to be followed by a remark. The remark is indicated by the use of an apostrophe (') preceeding it, and is terminated by either a colon (:) or the end of the line. For example:

```
100 I=1' INIT I:J=B*3'3 WORDS/ENTRY:GOSUB 234
'SETUP INFO
```

A statement consisting of just a remark is valid.

STOP — Similar to the END command except that the message BREAK @ LINE (x) is printed, where "x" is the line number of the STOP command.

USR — The USR command allows Basic to exit to a user provided assembly language routine, evaluate a value, and return with the result.

In use, Basic must be told where to go for the assembly language routine. When the USR function is referenced, Basic will call the USR transfer vector. Normally, this vector points to an error routine within Basic. In order to link to an assembly language routine, you must patch the address (start of Basic +6H) with a jump to your assembly language routine.

In your assembly language routine, in order to get the passed value, call 0027'H (start of Basic +27H). Basic will return with the passed value in registers D & E.

To return the result back to Basic, place the low byte of information in register B, and the high byte in register A, and call 002A'H (start of Basic plus 2AH).

To give control back to Basic, execute a RET instruction.

Having done the above, the Basic program and your routine can be made to interact at will by use of the USR function.

```
Example:  10 X=USR(Y)
          20 PRINT X
```

will pass the value "Y" to your assembly language routine. The returned value would be assigned to "X", and then printed on the console.

3.11 GROUP 11 COMMANDS TO HANDLE ASCII TEXT

ASAVE

The ASAVE command allows the punching of the ASCII text of a BASIC program. The format of the command is:

ASAVE

The command takes the entire current BASIC program and punches it in ASCII on the current punch device.

ALOAD,ALOAD*,AMERGE,AMERGE*

Four commands are used to allow the loading of ASCII source programs. These are ALOAD, ALOAD*, AMERGE, and AMERGE*. These commands all read ASCII text from the current reader device. Each incoming line must start with a line number, and terminate with a carriage-return. A line-feed immediately following the carriage-return is ignored, as are rubouts and nulls. Completely blank line (carriage return only) are also ignored. The input operation is terminated by either a control-Z in the text or by an EOF indication from the reader device. The two ALOAD commands clear the program storage area before starting, while the two MERGE commands merge the incoming lines with the existing program on a line number basic.

The difference between the Axxx commands and the Axxx* commands is the way in which they handle the reader device. The Axxx* commands assume a controlled reader, and stop after each line is read to convert the ASCII into internal format. The Axxx commands assume a non-controlled reader of any speed, and do not stop reading until the EOF is detected. These commands save the entire incoming ASCII text in memory prior to conversion to internal format, and hence require more memory for a given source than the Axxx* commands.

The format of the command is:

Axxx[*]

3.12 GROUP 12 SPECIAL FUNCTIONS & CONTROL-CHARACTERS

COMMA , Move to next TAB position or delimiter

SEMI-COLON ; 2 spaces between numbers

COLON : Used for multiple statements per line

RUBOUT The RUBOUT (DELETE) function echos the deleted characters bracketed by backslashes (\).

CONTROL-S,CONTROL-Q,CONTROL-C

During the execution of a BASIC program, the CTL-S and CTL-Q keys may be used to temporarily stop and then restart the program. The CTL-S key will stop the program, but will not echo or in any way affect any printed output. The CTL-Q key may then be used to resume the execution. A CTL-C may also be entered if it is desired to abort the execution and return to command level. Only CTL-Q and CTL-C will be recognized after a CTL-S.

CONTROL-U Delete input line.

CONTROL-X Return to the Monitor.

CONTROL-O Suppress the console output.

CONTROL-R A control-R entered whenever BASIC is accepting input (either while entering a program or entering data into a running program) will cause the current input buffer, with all rubouts and control-U's processed, to be typed out, and the input position to be left at the end of the line so more input may be entered. For example:

```
100 IFT\T\I-2\2-\=23 THEN GOSU\US\TO
123\32\32^R
```

would respond:

```
100 IF I=23 THEN GOTO 132
```

and leave the input pointer after the 132.

CONTROL-T While a program is running, a control-T may be entered on the console. This will result in the line number of the line currently being executed, being printed on the console. The program execution is unaffected by the use of this command.

3.13 GROUP 13 OPERATORS (listed in the order of evaluation)

A) Any expression enclosed in parentheses is evaluated from the innermost parenthesis first to the outermost parenthesis last.

B) ^ Exponentiation

C) (-) Negation. I.E. A minus sign placed so as to NOT indicate subtraction. For example:

```
A=-B or C=-(2*D)
```

D) * Symbol for multiplication. Used in the form 2*2(cr) yields an answer by Basic of "4".

E) / Symbol for division. Used in same manner as multiplication symbol.

F) + Symbol for addition. Example:

```
A+B     (add B to A)
```

G) - Symbol for substraction. Example:

```
A-B     (substract B from A)
```

H) RELATIONAL OPERATORS:

```
=    EQUALS
<>   NOT EQUAL
<    LESS THAN
>    GREATER THAN
<=   LESS THAN OR EQUAL TO
>=   GREATER THAN OR EQUAL TO
```

I) NOT Logical negation, such as A=NOT B

J) AND Logical AND

K) OR Logical OR

3.14 GROUP 14 ERROR HANDLING

ERL — Function used in error routine which allows it to process the error. This function requires no arguments (and hence no parentheses). The ERL function returns the line number at which the error occurred. A line number of 65535 indicates that the error occurred in a direct mode statement.

ERR — Function also used in error routine which allows it to process the error. This function requires no arguments (and hence no parentheses). The ERR function returns the number of the error which last occurred. A list of all defined error numbers is provided in Chapter 1, Section 1.3 of this manual.

ERROR — Software error generation. The ERROR command is provided to allow the use of software generated errors in conjunction with the error trapping capability. The format of the command is:

ERROR <error number>

where <error number> is any expression evaluating to an integer between 0 and 255. When this command is executed, an error occurs, and the value is stored as the error number. This number can either be one of the assigned error numbers (see list), or an arbitrary user defined number. If this command is executed, and no user error procedure is enabled (ON ERROR), then a normal program abort occurs. If the error number is defined, then the normal error message will be given. If not, the "UNKNOWN ERROR" message will be given.

ON ERROR GOTO — User error handling. The ON ERROR GOTO command is provided to specify a user error handling procedure. The command format is:

ON ERROR GOTO [<line number>]

where <line number> is the line number of the error handling routine. If the line number is omitted, then all user error trapping is disabled. After this statement is executed, any error occurring during the programs execution will cause a trap to the specified statement. This error routine has available two new functions, ERR and ERL, which allow it to process the error. See the descriptions of these functions under GROUP 14.

RESUME

After processing an error as required, the error routine returns to the regular program execution throught the RESUME statement. The format of this statement is:

RESUME [<line number>]

This statement is similar to the RETURN statement after a GOSUB, and in fact is nested in the same way. Every error trapping routine must eventually execute a RESUME statement. The RESUME statement with no line number re-executes the statement originally causing the error. The RESUME statement with a line number resumes execution at the specified line.

Since all error traps are nested in the same way as GOSUBS and function calls, it is possible for an error routine to begin with another ON ERROR statement, with its own error routine. In this case, each error routine must end by the execution of a RESUME statement.

It should be noted then when an error trap occurs, the effect of the ON ERROR statement which enabled the trap is disabled, and another error occurring prior to the execution of another ON ERROR statement will abort the program.

RESUME NEXT

This statement resumes execution at the statement (not the line) following the one causing the error. The format is: RESUME NEXT

CHAPTER 4

4.0 TDL BASIC VERSION 3 CAPABILITIES UNDER CP/M

A. Non-functional differences from Zapple version. The CP/M version of TDL Basic Version 3 works slightly different from the Zapple version due to the idiosyncracies of CP/M. For instance, CP/M always echos characters read from the console, which limits the interaction capabilities of programs. Specific differences are as follows:

1. The EDIT function always echos its commands. This may look slightly sloppy, but cannot be helped.

2. The rubout (DEL) function while typing in does not enclose the deleted input in backslashes (\), it merely echos the deleted characters.

3. The program break character is ctl-E rather than ctl-C. CP/M traps the ctl-C and exits to the operating system.

4. Basic V3 is loaded and executed by the command BASIC<cr>. A carriage return response to the "Memory Size?" message will properly assign available memory.

5. The ctl-X function is not available since there is no resident monitor in CP/M.

B. I/O differences. The CP/M version substitutes disk files for the Zapple reader/punch devices. All operations which could be performed to the reader or punch can now be performed to or from a currently assigned disk file. Only one input and one output disk file may be used at a time however.

C. Disk file assignment. Disk files may be dynamically assigned and deassigned to the input or output (reader or punch) function. This is done through the "OPEN" and "CLOSE" commands. The assignment is done through the following command:

OPEN #<file number>,<direction>,<file name>

For example:

OPEN #1,"I","PGM"

would open for reading a CP/M disk file called PGM.BAS.

In this, as in all following commands, the <file number> must be an expression which evaluates to 1, the only valid disk file number in this version. The <direction> must be a string expression which evaluates to either I or O (input or output), and the <file name> must be a string expression which evaluates to a standard CP/M file identifier with optional disk name and file extension. The disk name defaults to the logged in disk, and the file extension defaults to "BAS". This command assigns the input or output function to the specified disk file. It is an error to assign the input function to a non-existant disk file, or the output function to an existing one.

To deassign a file, the statement is:

CLOSE #<file number>,<direction>

Example:

CLOSE #1,"O"

For the input function, this merely breaks the file/function association. For the output function, all remaining memory buffers are emptied, and the file is closed, permanently establishing it on disk. If Basic V3 is aborted before an output file is closed, the file may be lost.

D. Disk file utilities. A number of additional facilities are available to maintain disk files in the CP/M version of Basic V3.

To determine if a particular file is on disk, the function "LOOKUP" is used. Its format is:

LOOKUP(<file name>)

Example:

LOOKUP "PGM.BAS"

The function returns a true (-1) if the file exists, and a false (0) if it does not.

The statement:

ERASE <file name>

Example:

ERASE "PGM.BAS"

erases the specified file from disk.

The statement:

RENAME <file name 1>, <file name 2>

Example:

RENAME "OLD.BAS","NEW.BAS"

renames file1 to file2.

E. Returning to CP/M. The command:

EXIT

closes any open files, and exits back to CP/M.

4.1 CP/M ERROR MESSAGES

29	DIRECTORY FULL
30	EXTENSION ERROR
31	NO DISK SPACE
32	INPUT FILE NOT FOUND
33	NO INPUT FILE
34	NO OUTPUT FILE
35	DUPLICATE OUTPUT FILE
36	OUTPUT CLOSE ERROR
37	INVALID OPEN TYPE
38	INVALID FILE ID