CHAPTER 10

PDOS BASIC COMMAND SUMMARY

The PDOS BASIC language is composed of commands, statements, operators, and functions. Commands are used to list, edit, save, load, execute, and debug your program. Commands begin with the command name and execute immediately. Statements begin with a line number and are used to perform a task or solve a problem. Operators perform operations on variables and are used within a statement. Likewise, functions are used within a statement and return specific values.

(CHAPTER 10 PDOS BASIC COMMAND SUMMARY continued)

(CHAPTER 10 PDOS BASIC COMMAND SUMMARY continued)

10.1 Function: ABS

Format: ABS <exp>
Definition: Return absolute value of <exp>

The ABS function returns the absolute value of <exp>.

```
LIST
 10  PRINT ABS[5],ABS[0],ABS[-100],ABS[-0.5]
 20  STOP
RUN
 5          0         100       0.5

STOP AT 20
```

10.2 Function: ADR

Format: ADR <exp>
Definition: Return memory address of <exp>

The ADR Function returns the memory address of the expression. This is useful in passing parameters to assembly subroutines or in accessing byte information within variables.

```
LIST
 10  MEM[ADR[A]]=72
 20  MEM[ADR[A]+1]=69
 30  MEM[ADR[A]+2]=76
 40  MEM[ADR[A]+3]=76
 50  MEM[ADR[A]+4]=79
 60  MEM[ADR[A]+5]=0
 70  PRINT $A;
RUN
HELLO

STOP AT 70
```

10.3 Statement: ALOAD

Format: ALOAD <string>,<exp1>,<exp2>
Definition: Load object file

The ALOAD statement uses the PDOS load file primitive (XLDF) to load a 9900 object module into memory. The file name is specified by <string>. The first expression <exp1> is the base address for the load operation. The maximum allowable length of the module (in bytes) is given by <exp2>.

```
LIST
 10  DIM A[100]: A=ADR A[0]
 20  ALOAD "GRAPH:OBJ",A,101*6
 30  EXTERNAL PLOT=A
 40  EXTERNAL COLOR=A+4
 50  COLOR=5
 60  PLOT=5,100,200
....
```

10.4 Operator: AND

Format: <exp1> AND <exp2>
Definition: Returns TRUE if <exp1> and <exp2> are nonzero

The Boolean operator AND compares two arithmetic expressions for nonzero values. If both are nonzero, the expression returns TRUE or 1. Otherwise, a zero is returned.

```
LIST
 10  I=1 AND 2: J=I AND 0
 20  IF I=1 AND J=0: PRINT "OK!"
 30  STOP
RUN
OK!

STOP AT 30
```

10.5 Function: ATN

Format: ATN <exp>
Definition: Return arctangent of radian <exp>

The ATN function returns the arctangent of the expression argument. The argument is given in radians.

```
LIST
 10  FOR I=1 TO 5
 20   PRINT I*ATN 1
 30  NEXT I
 40  STOP
RUN
 0.78539816
 1.5707963
 2.3561945
 3.1415927
 3.9269908

STOP AT 40
```

10.6 Statement: BASE

Format: BASE <exp>
Definition: Set CRU base for CRB and CRF functions

The BASE statement sets the CRU base value to <exp>. The base value is used by the BASIC functions CRB and CRF and is loaded into register R12 when the CRU functions are executed. The base remains unchanged until another BASE statement is executed.

See also 10.15 CRB and 10.16 CRF.

```
LIST
 10  BASE 040H
 20  FOR I=0 TO 4
 30   IF CRB[I]
 40     THEN PRINT "X ";
 50     ELSE PRINT "0 ";
 60  NEXT I
RUN
X X 0 X X
STOP AT 60
```

10.7 Statement: BAUD

```
   Format:     BAUD <exp1>,<exp2>{,<exp3>}
Definition:    Initialize TMS9902 user port <exp1> to
                  baud rate <exp2> and optionally set
                  CRU BASE and UNIT 2 to <exp3>
```

The BAUD statement initializes any one of the eight PDOS I/O ports and binds a physical TMS9902 UART to a character buffer. The command sets the 9902 character format, receiver and transmitter baud rates, and enables receiver interrupts.

```
BAUD 2,1200     Set aux port to 1200 baud
BAUD 3,9600     Set port 3 to 9600 baud
```

The first expression <exp1> selects the console port and ranges from 1 to 8. The system variable ITBCRU, located at address >0096 (>0086 for 102), points to the input CRU base table. This table binds a physical 9902 UART to a port character buffer and is generated during PDOS initialization. Entries in this table are changed by the BFIX utility or by the third expression, <exp3>, of the BAUD statement.

Port #1	= >0080	TM9900/101MA main port
2	= >0180	TM9900/101MA auxiliary port
3	= >0E00	ER3232 sel #1 page #0
4	= >0A00	ER3232 sel #3 page #0
5	= >0A40	ER3232 sel #3 page #1
6	= >0A80	ER3232 sel #3 page #2
7	= >0AC0	ER3232 sel #3 page #3
8	= >0B00	ER3232 sel #3 page #4

The TMS9902 UART's control register is initialized to 1 start bit, 7 bit character, even parity, and 2 stop bits (11 bits). The receiver and transmitter baud rates are initialized to the same value, according to expression <exp2>. The <exp2> expression ranges from 0 to 7 or the corresponding baud rates of 19200, 9600, 4800, 2400, 1200, 600, 300, and 110. Either parameter type is acceptable.

9902 initialized for 11 bits:
1 start bit
7 bit character
1 even parity
2 stop bits

<baud rate>	0	= 19200 baud
	1	= 9600 baud
	2	= 4800 baud
	3	= 2400 baud
	4	= 1200 baud
	5	= 600 baud
	6	= 300 baud
	7	= 110 baud

If a minus (-) precedes the port number, then the associated CRU base address is stored in the UNIT 2 (U2C(9)) variable. The third expression <exp3> is optional and binds a logical port to any 9902 UART CRU base.

See also 10.105 UNIT.

```
BAUD 3,0,0A40H     Set port 3 = >A40
                     ə 19200 baud
BAUD -3,9600       UNIT 2 = port 3
                     ə 9600 baud.
```

10.8a Function: BIT

Format: BIT[<var>,<exp>]
Definition: Returns <exp> bit of <var>

The BIT function returns the value of a specific variable bit. The variable name is specified by <var>, while <exp> specifies the bit displacement. The first bit number is 1.

```
LIST
 10 INPUT VAR
 20 FOR I=0 TO 47
 30 IF I-INT[I/16]*26=0: PRINT " ";
 40 PRINT #"0";BIT[VAR,I+1];
 50 NEXT I
 60 PRINT : GOTO 10
RUN
?1
 0000000000000000 0000000000000001 0000000000000000
?3.1415926
 0100000100110010 0100001111110110 1001101000100110
?-3283
 0000000000000000 1111001100101101 0000000000000000
?0.1
 0100000000011001 1001100110011001 1001100110011010


?_
```

10.8b Statement: BIT

Format: BIT[<var>,<exp1>]=<exp2>
Definition: Assign bit <exp1> of <var> a value of <exp2>

The BIT statement assigns a zero or one to any bit in a variable. <exp2> evaluates to zero or nonzero. <exp1> specifies the bit position within variable <var>. The first bit number is 1.

```
N=4*ATN 1
;N; 3.14159265
BIT[N,1]=1
;N; -3.14159265
```

10.9 Statement: BYE

Format: BYE
Definition: Return to PDOS monitor

The BYE statement exits to the PDOS monitor from BASIC. If no other program is run, BASIC can be entered again without destroying the old program. Resident PDOS user commands do not alter memory.

```
LIST
 10 REM PROGRAM HEADER
BYE
.SP 1
FREE=226,190
USED=455/476
.EX
*READY
LIST
 10 REM PROGRAM HEADER
_
```

10.10 Statement: CALL

Format: CALL <exp>...
CALL #<exp>

Definition: User defined function or assembly language call

The CALL statement evaluates all expressions separated by commas. This is particularly useful for calling user defined functions where the function value is not required.

If the first expression is preceded by a '#', then an assembly language routine is called. If <exp> evaluates to a number less than 256, then the value is used as an index into the BASIC CALL table for the subroutine address. If <exp> is greater than or equal to 256, then the value is used as the memory address of the user assembly language subroutine. By using the BASIC CALL table, programs do not have to be modified when using the standalone run modules.

The CALL table is located at memory address >2240. CALL #0 corresponds to the first entry in the CALL table (>2240), CALL #1 the second (>2242), and so forth.

BASIC communicates with the routine through the COM[] array. The address of COM[0] is passed to the subroutine in register R7.

The user subroutine call is via a Branch and Link (BL) instruction and hence register R11 contains the return address to the next line processor. Registers R7 through R11 must be preserved by the user subroutine!

Other BASIC system routines are accessible through register R11 as well. These are defined as follows:

B *R11 JMP NLIN. The BASIC interpreter continues executing on the same line as the CALL.

B @2(11) JMP LINE. The BASIC interpreter moves to the next line regardless of parameters or statements on the same line as the CALL.

BL @4(11) BL @EVAL. Evaluate the next expression in the CALL parameter list. Return address in register R2 and the delimiter in register R0. Only registers R5, R6, and R7 are preserved.

```
LIST
 100  PRINT "--- TOWER OF HANOI ---"
 110  INPUT "ENTER NUMBER OF DISKS",N
 120  INPUT "ENTER STARTING PEG",P
 130  INPUT "ENTER FINISHING PEG",R
 140  Q=6-P-R
 150  CALL FNMOVE[N,P,R,Q]
 160  STOP
 200  DEFN FNMOVE[N,P,R,Q]
 210  IF N=0: FNEND
 220  CALL FNMOVE[N-1,P,Q,R]
 230  PRINT "MOVE";P;" TO";R
 240  CALL FNMOVE[N-1,Q,R,P]
 250  FNEND
RUN
--- TOWER OF HANOI ---
ENTER NUMBER OF DISKS? 3
ENTER STARTING PEG? 1
ENTER FINISHING PEG? 3
MOVE 1 TO 3
MOVE 1 TO 2
MOVE 3 TO 2
MOVE 1 TO 3
MOVE 2 TO 1
MOVE 2 TO 3
MOVE 1 TO 3

STOP AT 160
```

Assembly routine:

```
       AORG >2240     ;BASIC CALL TABLE
       DATA >F000     ;CALL #0
       AORG >F000
>F000  INC @2(7)      ;INCREMENT COM[0]
       RT

LIST
 10  COM[0]=10
 20  CALL #0F000H
 30  PRINT COM[0]
 40  CALL #0
 50  PRINT COM[0]
RUN
 11
 12

STOP AT 50
```

(10.10 Statement: CALL continued)

BL @8(11) BL @EVSD. Examine the next parameter of the CALL list for a string variable or literal. Status returns HIGH if a string variable is found, LOW for a string literal, and EQUAL if neither is encountered. For strings, register R2 is returned pointing to the string and register R0 contains the delimiter. Only registers R5, R6, and R7 are preserved.

XOP <arg>,8 EVFIX <arg>. XOP 8 evaluates and fixes the next parameter of the CALL list to a 2's complement, 16-bit number. The <arg> can be any register (except R11) and the delimiter is returned in register R0. Like the other two calls, only registers R5, R6, and R7 are preserved.

For the three calls (BL @4(11), BL @8(11), and XOP <arg>,8), register R8 contains the program counter, R9 points to the task control block, and R10 is the stack pointer. The BASIC stack can be used for storing registers during execution of the subroutine. Parameters are pushed onto the stack by moving indirect R10 auto increment (MOV <arg>,*R10+), and popped from the stack by first decrementing R10 by two and moving the data off (DECT R10, MOV *R10,<arg>).

Delimiters are returned in register R0. They are left justified byte tokens and are defined as follows:

>00	<CR>	>0C	?
>01	'TO'	>0D	%
>02	'TAB'	>0E	\
>03	'STEP'	>0F	!
>04	'THEN'	>10	&
>05	'ELSE'	>11	.
>06	=	>12	[
>07	:	>13	]
>08	@	>14	"
>09	#	>15	'
>0A	,	>16	$
>0B	;		

Subroutine errors are reported to BASIC by executing the word >2EC0+error #. If the error # is greater than 31, then the word >2EE0 is executed, with the error # in the following word.

```
ERR05   DATA >2EC0+5    ;ERROR 5

ERR88   DATA >2EE0,88   ;ERROR 88
```

(10.10 Statement: CALL continued)

The following BASIC program illustrates the call procedure with the assembly subroutine to the right:

```
LIST
 10  DIM A[20],L[10]
 20  ALOAD "TEMP1",ADR A[0],21*6
 30  RECEIVE=ADR A[0]: SEND=RECEIVE+2
 100  CALL #SEND,SYS 36,E,"HOWDY DOODY"
 110  IF E: PRINT "SEND ERROR": STOP
 120  CALL #RECEIVE,T,$L[0]
 130  IF T<0
 140    THEN PRINT "NO MESSAGE"
 150    ELSE PRINT "TASK";T;" = ";$L[0]
 160  GOTO 100
RUN
TASK 0 = HOWDY DOODY
TASK 0 = HOWDY DOODY
TASK 0 = HOWDY DOODY
TASK 0 = HOWDY DOODY

ESCAPE AT 160
```

```
                        * SEND AND RECEIVE TASK MESSAGES
                        *
                        *      SEND=2,E,"HOWDY DOODY"
                        *      RECEIVE=T,$L[0]
                        *
          2E00                 DXOP EVFIX,8 ;EVALUATE & FIX
          2EC0          ERROR  EQU >2EC0    ;ERROR
                        *
          0004          EV     EQU 4        ;EVALUATE ADDRESS
          0008          ED     EQU 8        ;EVALUATE STRING
                        *
0000:     100F          RECV   JMP RECV2
                        *
0002:     C14B          SEND   MOV R11,R5   ;SAVE RETURN
0004:     2E06                 EVFIX R6     ;GET DEST TASK #
0006:     06A5 0004            BL @EV(5)    ;GET ERROR VAR
000A:     C1C2                 MOV R2,R7
000C:     04F7                 CLR *R7+     ;CLEAR 1ST WORD
000E:     04D7                 CLR *R7      ;DEFAULT=NO ERROR
0010:     06A5 0008            BL @ED(5)    ;GET STRING
0014:     1313                   JEQ ERR18  ;EXPECTING STRING
0016:     C006                 MOV R6,R0    ;TASK #
0018:     C042                 MOV R2,R1
001A:     2FDE                 XSTM         ;SEND TASK MESS
001C:     C5C0                   MOV R0,*R7 ;RETURN ERROR
001E:     0455                 B *R5        ;RETURN
                        *
0020:     C14B          RECV2  MOV R11,R5   ;SAVE RETURN
0022:     06A5 0004            BL @EV(5)    ;GET VAR ADDR
0026:     C182                 MOV R2,R6    ;SAVE ADDRESS
0028:     04F6                 CLR *R6+     ;CLEAR 1ST WORD
002A:     0716                 SETO *R6     ;DEFAULT=NO MESS
002C:     06A5 0008            BL @ED(5)    ;GET STRING
0030:     1206                   JLE ERR19  ;NONE OR STRING
0032:     C042                 MOV R2,R1    ;BUFFER ADDRESS
0034:     2FCB                 XGTM         ;GET TASK MESSAGE
0036:     1601                   JNE RECV4  ;NO MESSAGE
0038:     C580                 MOV R0,*R6   ;RETURN TASK #
                        *
003A:     0455          RECV4  B *R5        ;RETURN
                        *
003C:     2ED2          ERR18  DATA ERROR+18 ;EXPECTING STRING
003E:     2ED3          ERR19  DATA ERROR+19 ;EXPECTING STR-VAR
0040:          0000'           END RECV
```

10.11 Statement: CLEAR

    Format: CLEAR
Definition: Clear currently defined BASIC variables

The CLEAR statement clears all BASIC variables, stacks, and loop returns. This excludes the COM and MAIL arrays.

```
LIST
 10  DIM B[6]: B[3]=12: A=72
 20  $C="ABCD"
 30  PRINT A,B[3],$C
 40  CLEAR
 50  PRINT "A=";A
 60  PRINT "B=";B
 70  PRINT "$C=";$C
RUN
 72                12                ABCD
A= 0
B= 0
$C=

STOP AT 70
```

10.12 Statement: CLOSE

    Format: CLOSE <exp>
Definition: Close PDOS file by FILEID

The CLOSE statement closes the file specified by FILEID, <exp>. The FILEID is generated by the PDOS system on all file open statements and is used to subsequently reference the file.

If the file was opened for sequential access (OPEN or GOPEN) and the file was updated, then the END-OF-FILE marker is set at the current file pointer.

If the file was opened for random (ROPEN) or shared (SOPEN) access , then the END-OF-FILE marker is updated only if the file was extended -- that is, if data was written after the current END-OF-FILE marker.

The date of last update is adjusted in the disk directory only if the file has been altered.

All files must be closed after being opened! Otherwise, directory information is lost and possibly the file also.

```
LIST
 10  OPEN "TEMP",F
 20  PRINT "DISK/FILE SLOT=";F
 30  REM ....
 40  REM ....
 50  BINARY 1,F,1;3,C
 60  CLOSE F
RUN
DISK/FILE = 288

STOP AT 60
```

CLOSE ALL FILES!

10.13 Variable: COM

Format: COM[<exp>]

Definition: Common array not destroyed by NEW or RUN

The COM variable (referred to as the COMMON ARRAY) is a single dimensioned array which is used to preserve data during RUN, NEW, and program chaining. COM is initially dimensioned for ten elements, COM[0] through COM[9].

The size of the COM array is changed by assigning a new limit to SYS[8] and then executing a CLEAR or RUN statement. The new size remains until BASIC is executed again.

The COM array is used to pass and return parameters from assembly language subroutines. When a CALL is made to a subroutine, register R7 contains the address of COM[0].

See 10.10 CALL.

```
LIST
 10  DIM ARRAY[2]
 20  FOR I=0 TO 2
 30   ARRAY[I]=I+1: COM[I]=I+1
 40   PRINT ARRAY[I],COM[I]
 50  NEXT I
 60  STOP
RUN
 1               1
 2               2
 3               3

STOP AT 60
30
LIST
 10  DIM ARRAY[2]
 20  FOR I=0 TO 2
 40   PRINT ARRAY[I],COM[I]
 50  NEXT I
 60  STOP
RUN
 0               1
 0               2
 0               3

STOP AT 60


;SYS(8); 10
SIZE
PRGM:0
VNAM:0
VARS:0
FREE:31706
COM(9)=0
COM(10)=0
*ERROR 7
SYS(8)=20
CLEAR
SIZE
PRGM:0
VNAM:0
VARS:60
FREE:31646
COM(19)=0
COM(20)=0
*ERROR 7
```

10.14 Function: COS

Format: COS <exp>
Definition: Returns cosine of radian <exp>

The COS function returns the cosine of the angle <exp>, where <exp> is given in radians. One radian = approximately 57.29578 degrees (180/3.14159265).

The cosine is defined as the ratio of the length of the side adjacent to the angle <exp> to the length of the hypotenuse, in a right triangle.

```
LIST
 10  INPUT "ANGLE = ";A
 20  PRINT "COSINE OF";A;" DEGREES =";
 30  PRINT COS[A*0.0174533]
 40  PRINT "COSINE OF";A;" RADIANS =";
 50  PRINT COS[A]
 60  GOTO 10
RUN
ANGLE = 1
COSINE OF 1 DEGREES = 0.9998477
COSINE OF 1 RADIANS = 0.54030231
ANGLE = 3.14159265
COSINE OF 3.1415926 DEGREES = 0.99849715
COSINE OF 3.1415926 RADIANS = -1
ANGLE =
```

10.15a Function: CRB

Format: CRB <exp>
Definition: Returns CRU bit value of <exp> beyond BASE

The CRB function returns the value of a CRU bit displaced from the CRU base by <exp>.

See also 10.6 BASE.

```
LIST
 100  REM CHECK FOR AUX DSR
 110  BASE 00180H
 120  IF CRB[27]: PRINT "NO AUX DSR"
```

10.15b Statement: CRB

Format: CRB[<exp1>]=<exp2>
Definition: Loads CRU bit <exp1> beyond BASE with
  the Boolean value of <exp2>

The CRB statement executes a Set Bit Zero (SBZ) or Set Bit One (SBO), depending upon the Boolean value of <exp2>. The CRU bit affected is located at a displacement of <exp1> bits beyond the CRU base. The range of <exp1> is -128 to 127.

See also 10.6 BASE.

```
LIST
 100  REM RESET AUX 9902
 110  BASE 00180H
 120  CRB[31]=1
```

10.16a Function: CRF

Format: CRF <exp>

Definition: Return multiple CRU value of <exp> bits beyond BASE

The CRF function returns up to 16 bits of CRU data beginning at the BASE CRU address. <exp> specifies how many bits are to be read. The range of <exp> is 0 to 15, where 0 reads 16 bits.

See also 10.6 BASE.

```
LIST
 100 REM READ 12 BIT A/D VALUE
 110 BASE 00500H
 120 PRINT "A/D VALUE =";CRF[12]
```

10.16b Statement: CRF

Format: CRF[<exp1>]=<exp2>

Definition: Load multiple CRU value <exp2> into <exp1> bits at CRU BASE

The CRF statement outputs up to 16 bits of CRU data beginning at the BASE CRU address. <exp1> specifies how many bits are to be written. The range of <exp1> is 0 to 15, where 0 writes 16 bits. <exp2> is fixed to an integer. The lower 8 bits are output if <exp1> ranges from 1 to 8.

See also 10.6 BASE.

```
LIST
 100  REM SET AUX BAUD TO 600
 110  BASE 00180H
 120  CRB[31]=1  !RESET 9902
 130  CRF[8]=00062H  !SET CONTROL REGISTER
 140  CRB[13]=0  !FORGET INTERVAL TIMER
 150  CRB[12]=832  !SELECT 600 BAUD
```

10.17 Statement: DATA

```
    Format:     DATA <exp>,...,<string>,...
Definition:     Program data statements
```

A DATA statement contains data which is accessed by a READ statement. The items in the DATA statement are separated by commas and may include any expressions or strings. String literals are enclosed in single or double quotes.

See 10.76 READ and 10.80 RESTORE

```
LIST
 100  DIM A[10]
 110  READ I,$A[0]
 120  IF $A[0]="END": STOP
 130  PRINT I,$A[0]
 140  GOTO 110
 200  DATA 1,"ONE",2,"TWO",3,"THREE"
 210  DATA 4,"FOUR",5,"FIVE",6,"SIX"
 220  DATA 0,"END"
RUN
 1               ONE
 2               TWO
 3               THREE
 4               FOUR
 5               FIVE
 6               SIX

STOP AT 120
```

10.18 Statement: DATE

```
    Format:     DATE
                DATE <exp1>,<exp2>,<exp3>
                DATE <string-var>
Definition:     Read or set system date
```

The DATE statement reads, sets, or displays the system date.

DATE without any parameters displays to the user console an eight character string.

If the parameter of DATE is a string variable, then the same eight character string plus a null character is stored in the variable.

If expression <exp1> follows the DATE statement, it is evaluated and used to set the month of the system clock. A subsequent expression, <exp2>, sets the day, followed by an expression <exp3>, to set the year.

```
DATE
10/28/80
DATE 10,29,80
DATE $A[0]
;$A[0];10/29/80
```

10.19 Statement: DEFINE

    Format: DEFINE <string> {,<exp>}
Definition: Enter a file name in PDOS directory

The DEFINE statement creates in the disk directory a new file entry, as specified by <string>. A PDOS file name consists of an alpha character followed by up to 7 additional characters. An optional extension of up to 3 characters can be added if preceded by a colon. Likewise, the directory level and disk number are optionally specified by a semicolon and slash respectively.

If an expression follows the file name, then a contiguous file is allocated with length of <exp> sectors. This computes to 252 times <exp> bytes of data.

A contiguous file facilitates random access to file data since PDOS can directly position to any byte within the file without having to follow sector links. However, a contiguous file is changed to a non-contiguous file if it is extended past its initial allocation.

```
DEFINE "FILE;10"
FILES 10
DISK NAME=DISK #1/0                                  FILE
LEV  NAME:EXT       TYPE  SIZE    DATE CREATED       LAST

10   FILE           EX    0/1    05:25 10/29/80 05:23
*READY
```

10.20 Statement: DEFN

Format: DEFN FN<sim-var> {[<sim-var>,..]} = <exp>
Definition: Define a BASIC user runtime function

The DEFN statement allows the user to define new functions which are used the same as any intrinsic function. A user function is made up of either a single BASIC statement, or a multiple set of BASIC statements.

The user function name consists of the letters 'FN' followed by any simple variable name. (This name is a new entry in the symbol table). An optional parameter list may be included in the function definition. These parameters are referred to as dummy variables (local to the function definition) and must be enclosed in parentheses or brackets.

Single line user functions require an equal sign after the parameter list, followed by an arithmetic expression. Dummy parameters or any other global variable can be used in this expression. The function returns the result of the evaluated expression.

Multiple line user functions do not follow the parameter list with an equal sign. All program lines following the DEFN header, up to the first program line to begin with a FNEND statement, constitute the body of the function. Dummy variables are local to the body of a function. Additional local variables are declared with the LOCAL statement.

The value of the function is returned by assigning an expression to the function name. If no assignment is made within the body of the function, a zero is returned.

The function body need not be executed to be defined for program use. The RUN and CLEAR statements search and define all user functions before beginning execution.

If a function definition is encountered during execution, the body of the function is skipped and no statements is executed until after the first program line with the FNEND statement is found.

See 10.34 FNEND and 10.55 LOCAL.

```
LIST
 100  INPUT "DISTANCE=";X
 110  INPUT "MUZZLE VELOCITY=";V
 120  T=FNS[0,ATN 1]
 130  IF T<0: GOTO 100
 140  PRINT "ELEVATION IS";T*180/3.1415926;
 150  PRINT " DEGREES"
 160  PRINT X/(COS[T]*V);" SECONDS OF FLIGHT"
 170  GOTO 100

 500  DEFN FNA[A]=-9.8*X/(V*COS[A])+2*V*SIN[A]

 510  DEFN FNS[E1,E2]
 520  FOR I=1 TO 20
 530   II=(E1+E2)/2: FNS=II
 540   IF FNA[II]*FNA[E1]<=0: E2=II: GOTO 580
 550   IF FNA[II]*FNA[E2]>0
 560     THEN PRINT "NO SOLUTION": FNS=-1: FNEND
 570     ELSE E1=II
 580  NEXT I
 590  FNEND
RUN
DISTANCE=88167
MUZZLE VELOCITY=1000
ELEVATION IS 29.88646 DEGREES
 101.69034 SECONDS OF FLIGHT
DISTANCE=102040
MUZZLE VELOCITY=1000
ELEVATION IS 44.885373 DEGREES
 144.01851 SECONDS OF FLIGHT
DISTANCE=100
MUZZLE VELOCITY=1
NO SOLUTION
DISTANCE=
```

10.21 Statement: DELETE

Format: DELETE <string>
Definition: Delete a file from a PDOS disk

The DELETE statement removes from the disk directory, the file specified by <string>. All sectors associated with that file are also returned to the disk FREE space for use by other files on the same disk. A file cannot be deleted if it is delete or write protected. These protection flags must be cleared by a PDOS set file attributes command before the file can be deleted.

Since a bit map is maintained by PDOS for each sector, the deletion of files results in no loss of room on the disk nor is a disk compaction routine required to recover lost disk space. However, frequent file deletions and definitions creates small groups of contiguous sectors which tend to fracture files and make the creation of contiguous files impossible. This problem is easily remedied by periodically transferring all files to a newly initialized disk which would then allocate sectors sequentially for each file copied.

```
DEFINE "FILE;10"
FILES 10
DISK NAME=DISK #1/0                                FILE
LEV  NAME:EXT       TYPE    SIZE    DATE CREATED   LAST

10   FILE           EX       0/1   05:25 10/29/80 05:23
*READY
DELETE "FILE"
FILES 10
DISK NAME=DISK #1/0                                FILE
LEV  NAME:EXT       TYPE    SIZE    DATE CREATED   LAST

*READY
```

10.22 Statement: DIM

Format: DIM <dim-var>,...
Definition: Declare and allocate dimensioned variables

The DIM (DIMension) statement is used to define and allocate elements of a numeric or string array. An array can have up to 7 dimensions. Zero is always the first element of each dimension of any array.

Storage order has the right most dimension running the fastest. In other words, A[1,2] is stored as follows:

```
A[0,0] A[0,1] A[0,2] A[1,0] A[1,1] A[1,2]
```

An array dimensioned only once. Any attempt to reconfigure the dimension structure of an array is ignored or results in an error.

```
LIST
 10  DIM A[3,4,2]
 20  FOR K=1 TO 2: FOR I=1 TO 3: FOR J=1 TO 4
 30     A[I,J,K]=I
 40  NEXT J: NEXT I: NEXT K
 50  FOR K=1 TO 2: FOR I=1 TO 3: FOR J=1 TO 4
 60     PRINT A[I,J,K];
 70  NEXT J: PRINT: NEXT I: PRINT: NEXT K
RUN
 1 1 1 1
 2 2 2 2
 3 3 3 3

 1 1 1 1
 2 2 2 2
 3 3 3 3

STOP AT 90
```

10.23 Statement: DISPLAY

Format: DISPLAY <string>
Definition: List a PDOS file to console terminal

The DISPLAY statement displays on the user console, the disk file specified by <string>. The output is interrupted with the <escape> key. Since the output goes through the console routines, only TABs are expanded. Thus, files without line feeds print on one line.

DISPLAY is especially useful for displaying user screens that are stored on disk rather than in program memory. DISPLAY does a read only open; hence, other tasks may also be displaying the same file at the same time.

```
LIST
 100  DISPLAY "SCRN1"
 110  INPUT @[20,10];$NAM[0]
....
```

10.24 Statement: ELSE

Format: ELSE <statement>
Definition: A FALSE precondition to a line execution

The ELSE statement precedes any BASIC statement and continues execution of the program line only if the ELSE FLAG is FALSE. The ELSE FLAG is set FALSE whenever an IF statement is executed. If the IF statement evaluates true, the ELSE FLAG is set true. The flag remains set or reset until another IF statement is executed. Hence, multiple line blocks can be executed or ignored, depending upon what the IF evaluation returns.

During a LIST or LISTRP, the ELSE statement is indented by two blanks.

```
LIST
 100  INPUT "A=";A," B=";B
 110  IF A<B
 120   THEN PRINT "CONDITION TRUE"
 130   THEN PRINT A;" IS LESS THAN";B
 140   ELSE PRINT "CONDITION FALSE"
 150   ELSE PRINT A;" IS NOT LESS THAN";B
 160  GOTO 100
RUN
A=10 B=10
CONDITION FALSE
 10 IS NOT LESS THAN 10
A=1 B=2
CONDITION TRUE
 1 IS LESS THAN 2
A=
```

10.25 Statement: EQUATE

```
    Format:     EQUATE <sim-var>,<dim-var> {;...}
                       <single dim-var>,<exp> {;...}
Definition:     Assign simple variables to dimensioned
                variable entries
```

The EQUATE statement is used to equate simple variables to dimensioned variable elements. This makes dimensioned variable record elements more meaningful and reduces program storage.

The EQUATE statement is also used in passing arrays to functions. If the first parameter is a singly dimensioned array with index 0, then the expression <exp> is used as the array base address. The array name takes on the same array attributes as the passed parameter.

```
LIST
 100  DIM REC1[20]
 110  EQUATE NAME,REC1[0];PHONE,REC1[10]
 120  INPUT "NAME=";$NAME
 130  INPUT "PHONE=";PHONE
 140  PRINT $REC1[0]
 150  PRINT REC1[10]
      ....
RUN
NAME=JOHN DOE
PHONE=2242483
JOHN DOE
2242483
```

```
LIST
 10  DIM A[2,2]
 20  CALL FNFILL[A[0,0],10]: GOSUB PRINT
 30  CALL FNTRANS[A[0,0],2]: GOSUB PRINT
 40  STOP

 500  LABEL PRINT
 510  PRINT : FOR I=0 TO 2
 520   PRINT #" 990";A[I,0];A[I,1];A[I,2]
 530  NEXT I
 540  RETURN

 1000  DEFN FNTRANS[A,D]
 1010  EQUATE T[0],ADR[A]-8
 1020  FOR I=0 TO D: FOR J=I TO D
 1030    T=T[I,J]: T[I,J]=T[J,I]: T[J,I]=T
 1040   NEXT J: NEXT I
 1050  FNEND

 2000  DEFN FNFILL[A,R]
 2010  EQUATE T[0],ADR[A]-8
 2020  FOR I=0 TO 2: FOR J=0 TO 2
 2030    T[I,J]=INT[RND*R]
 2040   NEXT J: NEXT I
 2050  FNEND
RUN

   9   1   8
   3   2   9
   8   1   6

   9   3   8
   1   2   1
   8   9   6
```

10.26 Statement: ERROR

Format: ERROR <exp>
Definition: BASIC error trap routine at line <exp>

The ERROR statement designates a program line to which all execution errors trap. The transfer is done with the GOSUB routine. If an error occurs, the ERROR statement must be executed again for error trapping to continue. SYS[1] is set to the last error number, while SYS[2] contains the last line to have an error.

```
LIST
 100  REM GET LARGE AND SMALL #
 110  ERROR 150
 120  I=10
 130  J=1/I: I=I/10: GOTO 130
 150  IF SYS[1]=28: PRINT "DIVISION BY ZERO"
 160  IF SYS[1]=29: PRINT "OVERFLOW"
 170  PRINT " AT LINE";SYS[2]
 180  PRINT "LARGE=";J
 190  PRINT "SMALL=";I
RUN
OVERFLOW AT LINE 130
LARGE= 1E75
SMALL= 1E-76

STOP AT 190
```

10.27 Statement: ESCAPE

Format: ESCAPE
Definition: Allow the ESC key to break execution

The ESCAPE statement enables the <escape> key to break program execution. ESCAPE has no effect in keyboard mode.

```
LIST
 1000  NOESC  !NO BREAK ALLOWED UNTIL COMPLETED
 1010  FOR I=1 TO 1000
 1020   SAMPLE[I]=MEMW[0E300H]
 1030  NEXT I
 1040  ESCAPE  !ALLOW BREAK AGAIN
```

10.28 Statement: EVENT

Format: EVENT <exp>
Definition: Set or reset event flag bit

The EVENT statement sets or resets an event flag bit. The expression <exp> specifies both the event number and its value. If <exp> is positive, then the event bit is set to 1. If <exp> is negative, the event is reset to 0. A hardware event can be simulated with the EVENT statement by setting an event of 1 through 15.

See 5.2.14 XSEF - SET EVENT FLAG

```
100  MAIL[2]=VALUE
110  EVENT 30  !SIGNAL MAIL READY

200  NVAL=MAIL[2]
210  EVENT -30  !SIGNAL MAIL RECEIVED
```

10.29 Function: EVF

Format: EVF <exp>
Definition: Test event flag

The test event flag function EVF returns a 0 or 1, depending upon the value of the event bit specified by <exp>. The event flag is not altered by the function. The event number if given by the expression modulo 128.

See 5.2.18 XTEF - TEST EVENT FLAG

```
100  EVENT 30  !SIGNAL READY
110  REM WAIT FOR REPLY
120  IF EVF[30]: SWAP : GOTO 110
```

10.30 Function: EXP

Format: EXP <exp>

Definition: Returns e raised to the <exp> power

The EXP function returns the exponentiation of <exp>. This is defined as e (2.71828...) raised to the power of <exp>. The exponential function is the inverse of the LOG function.

```
LIST
 10  FOR I=0.5 TO 2 STEP 0.1
 20   PRINT EXP I; TAB EXP[I]*7;"*"
 30  NEXT I
RUN
 1.6487213 *
 1.8221188  *
 2.0137527    *
 2.2255409     *
 2.4596031       *
 2.7182818         *
 3.004166            *
 3.3201169             *
 3.6692967               *
 4.0552                     *
 4.4816891                     *
 4.9530324                        *
 5.4739474                            *
 6.0496475
 6.6858944
 7.3890561

STOP AT 30
;EXP(LOG(4.5)); 4.5
```

10.31 Statement: EXTERNAL

Format: EXTERNAL <sim-var>=<exp>

Definition: Define external subroutine call

The EXTERNAL statement places an entry in the external table and defines an external variable. The external variable <sim-var> is then used to call the external subroutine at address <exp>.

If <sim-var> has already been defined, no entry is made. Hence, EXTERNAL statements should be executed at the beginning of the program.

Once the call has been made, all parameters, links, and register usage are identical to those of the CALL statement.

The external table size defaults to 20 entries. The size is changed by assigning SYS[34] the new size and then executing a RUN or CLEAR statement. SYS[35] points to the external table.

See 10.10 CALL.

```
 100  DIM A[700]: A=ADR A[0]
 110  ALOAD "EXCOM",A,700*6
 120  EXTERNAL SPEAK=A
 122  EXTERNAL COLOR=A+4
 124  EXTERNAL MODE=A+8
 126  EXTERNAL MOVE=A+12
 128  EXTERNAL PATTERN=A+16
 130  EXTERNAL PLOT=A+20
 132  EXTERNAL SPRITE=A+24
 134  EXTERNAL CIRCLE=A+28
 136  EXTERNAL SOUND=A+32
 138  EXTERNAL ADC=A+36
 140  EXTERNAL APU=A+40
 160  SOUND=0
 200  MODE=4,1;-1
 210  SPRITE=0,"0000183C7E7E3C18"
 220  COLOR=4
 ....
```

10.32 Statement: FILE

```
   Format:     FILE <exp>...
Definition:    Select, read, write, position, lock,
                  or unlock file
```

The FILE statement is the primary file I/O statement and is used to SELECT, READ, WRITE, POSITION, LOCK, or UNLOCK a file. The first expression selects the FILE command, and any additional parameters follow. Multiple FILE functions can be placed in one statement by using a semicolon to precede the new function.

```
LIST
 10 SELECT=1  !FILE SELECT
 20 WRITE=2  !FILE WRITE
 30 READF=3  !FILE READ
 40 POSITION=4  !FILE POSITION
 100 OPEN "#TEMP",F
 110 FOR I=0 TO 500
 120  FILE SELECT,F;WRITE,I,I*I,I*I*I
 130 NEXT I
 140 CLOSE F
 200 ROPEN "TEMP",F
 210 I=INT[RND*500]
 220 FILE SELECT,F;POSITION,18,I,0
 230 FILE READF,J,K,L
 240 IF I<>J: PRINT "ENTRY";I;" READ AS";J;K;L
 250 PRINT I,J;K;L
 260 GOTO 210
RUN
 362            362 131044 47437928
 5              5 25 125
 326            326 106276 34645976
 119            119 14161 1685159
 182            182 33124 6028568
 11             11 121 1331
 484            484 234256 113379900
 48             48 2304 110592
 ....
```

0) SELECT and LOCK TASK

```
FILE 0,<exp1>,{<exp2>}

exp1 = file slot ID

exp2 = number of bytes per variable.  This is
       an optional parameter and applies only
       to variables within one FILE heading.
       Default is 6 bytes.
```

The task remains locked until the entire FILE command is executed. FILE 0 is used when two users are randomly accessing the same file.

1) SELECT FILE

```
FILE 1,<exp1>,{<exp2>}

exp1 = file slot ID

exp2 = number of bytes per variable.  This is
       an optional parameter and applies only
       to variables within one FILE heading.
       Default is 6 bytes.
```

```
ROPEN "FILE",FILID
FILE 1,FILID
```

2) WRITE TO FILE

```
FILE 2,<exp>....

exp = A list of variables to be written to
      the file.
```

```
FILE 2,1,A,N[0],N[1]
```

(10.33 Statement: FILE continued)

3) READ FROM FILE

```
FILE 3,<var>....
```

var = A list of variables which receive data read from a file.

```
FILE 3,A,B,N[0],N[1]
```

4) POSITION FILE

```
FILE 4,<exp>
```

exp = A single byte index into a ROPENed file. This parameter can be larger than 32767.

```
FILE 4,RECN*RECL
```

```
FILE 4,<exp1>,<exp2>,<exp3>
```

exp1 = record length in bytes
exp2 = record number
exp3 = byte displacement into record

File index = exp1 x exp2 + exp3

*No expression can exceed 32767.

```
FILE 4,4*6,I,0
```

5) WRITE LINE

```
FILE 5,<string>....
```

string = String to be written to the file. String is delimited by a null character. A <carriage return> is not appended to the end of the string.

```
FILE 5,"HELLO TURKEY"
```

6) READ LINE

```
FILE 6,<string-var>....
```

string-var = String variable into which a line of characters is read. A line is defined as a string which is less than 132 characters long and delimited by a carriage return. The <CR> is replaced by a null and <LF>'s are dropped.

```
LIST
 10  DIM A[20]
 20  OPEN "LIST",F
 30  FILE 1,F;6,$L[0]
 40  PRINT $L[0]
 50  GOTO 30
```

(10.33 Statement: FILE continued)

FILE 5 is the complement of FILE 6. However, FILE 5 writes characters until a null character is found while, FILE 6 reads until a <carriage return> is found. Hence, if a FILE 5 line is to be read by a FILE 6, then a <carriage return> must first be appended to the line.

```
LIST
 100  DIM A[10]
 110  $A[0]="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
 120  $CR=%13%0
 130  ROPEN "TEMP",F
 140  FOR I=1 TO 5
 150   FILE 1,F;5,$A[0],$CR
 160  NEXT I
 170  FILE 1,F;4,0
 180  FOR I=1 TO 5
 190   FILE 1,F;6,$A[0]
 200   PRINT $A[0]
 210  NEXT I
 220  CLOSE F
RUN

ABCDEFGHIJKMLNOPQRSTUVWXYZ
ABCDEFGHIJKMLNOPQRSTUVWXYZ
ABCDEFGHIJKMLNOPQRSTUVWXYZ
ABCDEFGHIJKMLNOPQRSTUVWXYZ
ABCDEFGHIJKMLNOPQRSTUVWXYZ
STOP AT 220
```

7) LOCK FILE

```
FILE 7,<exp>,<var>

<exp> = file slot ID
<var> = error return variable
```

The FILE 7 statement prevents access to a shared file by any other task. The expression <exp> specifies the file by FILE ID. The variable <var> is returned with a zero if the lock is successful. Otherwise, the error number is returned. Possible error numbers include:

52 = File not open
59 = Invalid file slot
75 = File locked

```
LIST
 10  SOPEN "DATAF",FILID
 20  FILE 7,LOCK FILID,ER: IF ER: GOTO 20
 30  FILE 1,FILID;4,0;3,A
 40  A=A+1
 50  FILE 4,0;2,A
 60  FILE 8,FILID
```

(10.33 Statement: FILE continued)

8) UNLOCK FILE

FILE 8,<exp>

<exp> = file slot ID

The FILE 8 statement unlocks a locked shared file for access by other tasks.

```
LIST
 10  SOPEN "FILE2",F
 20  FILE 7,F,E  !LOCK FILE
 30  REM PROCESS RECORD
 ....
 90  FILE 8,F  !UNLOCK FILE
```

The FILE 0 and 1 file selection remains valid for all subsequent READs and WRITEs until another FILE 0 or 1 is executed. However, the variable size option of the FILE 1 statement is valid only for the FILE statement in which it was executed. Thus, a FILE 1 command is required with a semicolon specifying another FILE command, in order to use this optional parameter.

There is no end of file test. An ERROR trap is required to detect any file errors.

```
LIST
 10  ERROR 100
 20  OPEN "LIST",F
 30  FILE 1,F,1;3,C
 40  PRINT $C;
 50  GOTO 30
 100  POP: CLOSE F
 110  STOP
```

10.33 Command: FILES

```
    Format:     FILES <list>
Definition:     List PDOS directory
```

The FILES command sends the <list> string to PDOS for directory file listings. The <list> parameter selects file type, directory level, and/or disk. The syntax is:

```
        FILES {file type}{protection}{level qualifier}{/disk #}

         {file type} =  AC  Assign Console file
                        BN  Binary file
                        BX  PDOS BASIC token file
                        EX  PDOS BASIC file
                        OB  TI9900 object file
                        SY  System file
                        TX  Text file
                        UD  User defined

        {protection} =  *   Delete protected
                        **  Delete and write protected

   {level qualifier} =  #   List all files on level #
                        @   List all files

           {/disk #} =  disk number, ranging from 0 to 127
```

Example:

```
        FILES @/1
        DISK NAME=PAUL #30MD/1                              FILES=17/64
        LEV  NAME:EXT      TYPE     SIZE     DATE CREATED   LAST UPDATE

         1   SYFILE:SR               7/19   04:00 02/26/81  20:28 02/26/81
         1   ASM           SY       43/43   09:50 02/27/81  09:51 02/27/81
         1   JEDY          SY       25/25   09:51 02/27/81  09:51 02/27/81
         1   SYFILE        OB        3/4    20:14 02/26/81  20:28 02/26/81
         1   PLIST:SR               41/41   15:42 02/27/81  15:42 02/27/81
         1   LIST                    5/40   12:17 03/01/81  11:22 03/07/81
         0   $TTA                    1/1    10:01 03/01/81  10:01 03/01/81
         0   $TTA1                   1/1    10:01 03/01/81  10:01 03/01/81
         1   DOC                     0/1    05:06 02/01/81  05:06 02/01/81
         1   LOAD:SR                 4/4    05:14 02/01/81  10:02 03/02/81
         1   LOAD          OB        2/2    05:14 02/01/81  10:02 03/02/81
         1   DFIX          SY        2/2    13:03 03/03/81  13:03 03/03/81
         1   PRINTS        EX       19/22   15:37 03/05/81  04:27 03/06/81
         1   NYM           OB       21/21   22:07 03/05/81  22:07 03/05/81
         1   BURN302:SR             68/68   22:22 03/05/81  22:23 03/05/81
         1   BURN302       OB       28/28   22:23 03/05/81  22:24 03/05/81
         1   TEMP                    5/5    10:47 03/08/81  10:47 03/08/81
        *READY
```

10.34 Statement: FNEND

Format: FNEND

Definition: End of a user defined function

The FNEND statement is used to terminate the body of a multi-line function when it immediately follows the program line number. It also causes the program to exit from a function during execution. Hence, the FNEND can appear anywhere within the function body, but at the beginning of a line, FNEND terminates the function definition.

```
LIST
 100  INPUT "REAL=";R;" IMAG=";I
 120  PRINT "COMPLEX MODULUS=";FNCMOD[R,I]
 130  GOTO 100

 500  DEFN FNCMOD[REAL,IMAG]
 510  LOCAL I,J
 520  I=ABS REAL: J=ABS IMAG
 530  IF I=J: FNCMOD=I*SQR 2: FNEND
 540  IF I<J: FNCMOD=J*SQR[1+I*I/(J*J)]: FNEND
 550  FNCMOD=I*SQR[1+J*J/(I*I)]
 560  FNEND
RUN
REAL=2 IMAG=2
COMPLEX MODULUS= 2.8284271
REAL=3 IMAG=-4
COMPLEX MODULUS= 5
REAL=
```

10.35 Statement: FOR

Format: FOR <sim-var>=<exp1> TO <exp2> {STEP <exp3>}
Definition: Header for a BASIC loop

The FOR and NEXT statements indicate the start and end of an instruction block that is to be repeatedly executed. The <sim-var> is the control variable and is initialized to <exp1> when the FOR statement is executed. This variable is incremented (or decremented) by <exp3> when the corresponding NEXT statement is executed. If no STEP is specified, a default step value of 1 is used.

After the control variable has been updated by the NEXT statement, it is compared with <exp2>. If it is greater than <exp2> and the STEP value is positive, or if it is less than <exp2> and the STEP value is negative, then the loop is terminated and execution continues after the NEXT statement. Otherwise, execution returns to the FOR statement for another pass.

A pre-check is made by the FOR statement to see if the termination value has already been achieved. If such is the case, BASIC searches forward for the corresponding NEXT statement and the loop sequence is not executed. (The corresponding NEXT statement must be the first statement of the program line for this to work properly.)

The control variable is often used in the computations within the instruction block. It may be changed within the body of the loop and the latest value of the variable is used in the exit test.

It is possible for the FOR and NEXT statements to be on the same program line. However, this type of a loop structure cannot be interrupted by the escape key. Also, as stated above, use of FOR and NEXT statements on the same line result in an error if, during the pre-check, the loop is terminated.

FOR loops may be nested. However, they should not use the same control variable and loops must be completely contained within the other. Overlapping is not permitted. Inner loops run to completion before outer loops. PDOS BASIC allows up to eight levels of FOR/NEXT nested loops.

```
LIST
 10  FOR I=1 TO 5 STEP 2
 20   PRINT I;
 30  NEXT I
 40  PRINT: PRINT I
RUN
 1 3 5
 7

STOP AT 40
```

```
LIST
 10 FOR I=2 TO -6 STEP -2
 20  PRINT I;
 30 NEXT I
 40 PRINT: PRINT I
RUN
2 0 -2 -4 -6
-8

STOP AT 40
```

```
LIST
 10  FOR I=1 TO 4
 20   FOR J=1 TO 10
 30     PRINT #"990";I*J;
 40   NEXT J
 50   PRINT
 60  NEXT I
RUN
  1  2  3  4  5  6  7  8  9 10
  2  4  6  8 10 12 14 16 18 20
  3  6  9 12 15 18 21 24 27 30
  4  8 12 16 20 24 28 32 36 40

STOP AT 60
```

(10.35 Statement: FOR continued)

Program transfers out of the loop are permitted, but transfers into the loop are not, except for the purpose of completing an existing loop structure. Of course, a subroutine call is permissible since it returns for proper loop termination.

Every FOR statement causes the subsequent program statements to be indented by one character when the program is LISTed. This is accumulative. The NEXT statement conversely decrements this indentation count by one.

FOR loops can be nested up to 8 levels deep. You change this value with SYS[14]. Assign the new depth to SYS[14] and then execute a CLEAR or RUN statement.

```
LIST
 10  FOR I1=1 TO 10
 20   FOR I2=1 TO 10
 30    FOR I3=1 TO 10
 40     FOR I4=1 TO 10
 50      FOR I5=1 TO 10
 60       FOR I6=1 TO 10
 70        FOR I7=1 TO 10
 80         FOR I8=1 TO 10
 90          FOR I9=1 TO 10
 100           FOR I10=1 TO 10
 110            FOR I11=1 TO 10
 120             FOR I12=1 TO 10
RUN

*ERROR 11 AT 90
;SYS(14); 8
SYS(14)=12
RUN

STOP AT 120
```

10.36 Statement: FNPOP

Format: FNPOP
Definition: Pop function call from system heap

The FNPOP statement pops a function call from the system heap. Functions must be gracefully exited! Variable addresses and pointers are stored on the system heap and must be restored in an orderly manner.

```
LIST
 10  INPUT I
 20  PRINT I;" FACTORIAL=";FNFACT[I]
 30  GOTO 10

 100  DEFN FNFACT[I]
 110  ERROR FERR
 120  IF I<=1: FNFACT=1: FNEND
 130  FNFACT=I*FNFACT[I-1]
 140  FNEND

 200  LABEL FERR
 210  POP : PRINT "ERROR"
 220  IF SYS[32]: FNPOP : GOTO 220
 230  GOTO 10
RUN
? 6
 6 FACTORIAL= 720
? 10
 10 FACTORIAL= 3628800
? 50
 50 FACTORIAL= 3.0414093E64
? 100
 100 FACTORIAL=ERROR
? 10
 10 FACTORIAL= 3628800
? 100
 100 FACTORIAL=ERROR
? 50
 50 FACTORIAL= 3.0414093E64
?
```

10.37 Function: FRA

Format: FRA <exp>
Definition: Returns fractional part of <exp>

The FRA function returns the fractional part of <exp>.

```
LIST
 10  A=-1
 20  FOR I=0 TO 47
 30   IF FRA[I/16]=0: PRINT " ";
 40   PRINT #"0";BIT[A,I+1];
 50  NEXT I
RUN
 0000000000000000 1111111111111111 0000000000000000
STOP AT 50
```

10.38 Statement: FREE

Format: FREE <exp>
Definition: Free or reclaim task memory

The FREE statement frees or reclaims memory from the highest memory address of the task. Variable definitions, and GOSUB and FOR/NEXT stack addresses, are adjusted accordingly. If <exp> is positive, memory is freed. If <exp> is negative, memory is reclaimed.

This statement is very useful in creating global data areas, spawning new tasks, or passing storage to assembly language routines.

A BASIC subroutine that uses the FREE statement in spawning a new task is shown below:

```
2000  REM CREATE TASK
2010  DIM C[5],L[10]
2020  FREE 1024  !FREE 1k
2030  $L[0]="LT.LS 10.KT 0"
2040  COM[0]=ADR[L[0]]  !COMMAND LINE
2050  COM[1]=SYS[28]  !LOW ADDRESS
2060  COM[2]=SYS[29]  !HIGH ADDRESS
2070  COM[3]=1  !TASK TIME
2080  COM[4]=SYS[10]  !CRT PORT
2090  $C[0]=%"05C70700C057 C0A70012C0E7"
2100  $C[2]=%"0018C1270006 C167000C0407"
2110  $C[4]=%"2FDDC5C0C9C0 0006045B"
2120  CALL #ADR CREATE[0]  !CREATE TASK
2130  IF COM[0]
2140    THEN PRINT "PDOS ERROR";COM[0]
2150    THEN GOTO 2170
2160    ELSE IF TSK[COM[1]]>0: GOTO 2130
2170  FREE -1024  !RECOVER SPACE
2180  RETURN
```

```
LIST
 100  DIM L[10]
 110  GOSUB 500
 120  FREE 4096  !FREE 4K
 130  GOSUB 500
 140  FREE -4096  !RECOVER 4K
 150  STOP
 500  RESTORE : PRINT
 510  FOR I=20 TO 29
 520   READ $L[0]: PRINT  TAB 26-LEN L[0];
 530   PRINT $L[0];"=";#SYS[I]
 540  NEXT I
 550  RETURN
 600  DATA "BEGINNING USER PROGRAM"
 610  DATA "STATEMENT DEFINITION TABLE"
 620  DATA "VARIABLE NAME TABLE"
 630  DATA "VARIABLE DEFINITION TABLE"
 640  DATA "NEXT VARIABLE DEFINITION"
 650  DATA "NEXT VARIABLE STORAGE"
 660  DATA "GOSUB STACK"
 670  DATA "FOR/NEXT STACK"
 680  DATA "END USER STORAGE"
 690  DATA "END TASK MEMORY"
RUN

    BEGINNING USER PROGRAM=62F2
STATEMENT DEFINITION TABLE=644A
       VARIABLE NAME TABLE=64A4
 VARIABLE DEFINITION TABLE=64B2
  NEXT VARIABLE DEFINITION=64BA
     NEXT VARIABLE STORAGE=DE92
               GOSUB STACK=DF1E
            FOR/NEXT STACK=DF6E
          END USER STORAGE=E000
           END TASK MEMORY=E000

    BEGINNING USER PROGRAM=62F2
STATEMENT DEFINITION TABLE=644A
       VARIABLE NAME TABLE=64A4
 VARIABLE DEFINITION TABLE=64B2
  NEXT VARIABLE DEFINITION=64BA
     NEXT VARIABLE STORAGE=CE92
               GOSUB STACK=CF1E
            FOR/NEXT STACK=CF6E
          END USER STORAGE=D000
           END TASK MEMORY=E000
```

10.39 Statement: GLOBAL

Format: GLOBAL <exp1>,{<var>...}
Definition: Declare common variable storage

The GLOBAL statement defines all variables listed after the first expression <exp1>, beginning with the address <exp1>. This is used in creating a common variable area that can be shared with other tasks.

Variables are assigned only if previously undimensioned. Hence, GLOBAL should be one of the first statements in a program. Other program tasks should use the exact same GLOBAL statement so that storage allocation is the same.

```
LIST                        {task 0 program}
 100  DIM CM[70]  !GLOBAL WORK AREA
 110  MAIL[0]=ADR CM[0]  !PASS TO OTHER TASKS
 120  GLOBAL MAIL[0],A,B[10],C[10,4],VEL
 ....
```

```
LIST                        {task 1 program}
 100  IF MAIL[0]=0
 110    THEN GOTO 100  !WAIT FOR ADDRESS
 120  GLOBAL MAIL[0],A,B[10],C[10,4],VEL
 ....
```

10.40 Statement: GOPEN

Format: GOPEN <string>,<var>
Definition: Open a PDOS file read only access

The GOPEN statement opens the file <string> in read only mode for PDOS BASIC file access. The FILE ID is returned in <var>. Thereafter, the file is referenced by the FILE ID and not by name.

Since the file cannot be altered, it cannot be extended, nor is the LAST UPDATE parameter be changed when it is closed. All data transfers are buffered through a channel buffer.

```
LIST
 10  DIM NAME[10]
 20  INPUT "FILE NAME=";$NAME[0]
 30  GOPEN $NAME[0],FILEID
 40  ERROR 100
 50  COUNT=0
 60  BINARY 1,FILEID,1;3,I: COUNT=COUNT+1
 70  GOTO 60
 100  POP: RESET
 110  PRINT "FILE LENGTH=";COUNT;" BYTES"
 120  GOTO 20
RUN
FILE NAME=PRGM1
FILE LENGTH= 546 BYTES
FILE NAME=_
```

10.41 Statement: GOSUB

Format: GOSUB <exp>
Definition: BASIC subroutine call

The GOSUB statement is used to branch out of a program sequence to a BASIC subroutine. The GOSUB statement pushes the address of the statement immediately following the GOSUB statement onto the the GOSUB stack and passes execution to the line number <exp>.

A RETURN statement is used to exit the subroutine and resume execution at the first statement following the GOSUB statement. This pops the top of the GOSUB stack. All subroutines should exit via a RETURN statement so that the top address is removed from the GOSUB stack.

Subroutines may be nested up to 20 levels. The maximum nesting depth is altered by assigning a new size to SYS[15] and executing a CLEAR or RUN command.

Executing a RETURN statement when no previous GOSUB statement has been executed results in an error.

```
LIST
 10  DIM STCK[50]
 20  INPUT "NUMBER=";N;
 30  GOSUB FACTORIAL
 40  PRINT " FACTORIAL=";R
 50  GOTO 20

 100  LABEL FACTORIAL
 110  I=0
 120  IF N<=1: R=1: RETURN
 130  STCK[I]=N: N=N-1: I=I+1
 140  GOSUB 120
 150  I=I-1: N=STCK[I]
 160  R=R*N: RETURN
RUN
NUMBER=4 FACTORIAL= 24
NUMBER=10 FACTORIAL= 3628800
NUMBER=20 FACTORIAL= 2.432902E18
NUMBER=21
*ERROR 11 AT 140

SYS(15)=40
RUN
NUMBER=20 FACTORIAL= 2.432902E18
NUMBER=21 FACTORIAL= 5.1090942E19
NUMBER=30 FACTORIAL= 2.6525286E32
NUMBER=40 FACTORIAL= 8.1591528E47
NUMBER=41
*ERROR 11 AT 140
```

10.42 Statement: GOTO

Format: GOTO <line #>
Definition: Unconditional program transfer

The GOTO statement does an unconditional program transfer to the line number specified by <line #>.

```
LIST
 10  INPUT X
 20  IF X=0: GOTO 50
 30  PRINT "X IS NOT ZERO"
 40  GOTO 10
 50  PRINT "X IS ZERO"
 60 GOTO 10
RUN
? 0
X IS ZERO
? 1
X IS NOT ZERO
? _
```

10.43 Statement: IF

Format: IF <logical exp> {: <statement>}
Definition: Conditional preprocessor for a command

The IF statement is used to set the ELSE FLAG. At the beginning of the IF statement, the flag is set FALSE. If the <logical exp> evaluates true, the flag is set TRUE. The THEN statement executes on a TRUE flag, while the ELSE statement executes on a FALSE flag.

Additional statements can follow the IF statement on the same line and execute on the TRUE condition. These statements are separated from the <logical exp> by a colon.

The <logical exp> is any one of the following types:

```
<exp>
<exp> <relation> <exp>
<string>
<string> <relation> <string>
<string> <relation> <string> , <exp>
```

An <exp> alone evaluates TRUE if nonzero and FALSE if zero. The same applies to <string> alone.

If a <string> <relation> <string> is followed by an <exp>, then the two strings are compared for only <exp> characters.

```
LIST
 10  READ A,B,C
 20  IF C=0: STOP
 30  FLAG=0
 40  IF A+B<C: FLAG=1
 50   ELSE IF A+C<B: FLAG=1
 60   ELSE IF B+C<A: FLAG=1
 70  IF FLAG
 80   THEN $Y="NOT "
 90   ELSE $Y=""
 100  PRINT "SIDES";A;B;C;
 110  PRINT " ARE ";$Y;"A TRIANGLE"
 120  GOTO 10
 130  DATA 3,4,5,3,3,9,8,5,1,3,1,0
RUN
SIDES 3 4 5 ARE A TRIANGLE
SIDES 3 3 9 ARE NOT A TRIANGLE
SIDES 8 5 1 ARE NOT A TRIANGLE

STOP AT 20
```

10.44 Function: INP

Format: INP <exp>
Definition: Returns integer part of <exp>

The INP function returns the integer part of <exp>. It also guarantees the result to be in integer format. That is, the first word zero is followed by the 16-bit 2's complement value.

The range of <exp> is -32767 to 32767. For larger values, the INT function must be used.

```
LIST
 10  INPUT "N=";N;
 20  PRINT TAB 15;"INP[N]=";INP[N]
 30  GOTO 10
RUN
N=10.2         INP[N]= 10
N=-534.345     INP[N]= -534
N=33000.5      INP[N]=
*ERROR 30 AT 20
```

10.45 Statement: INPUT

Format: INPUT <input list>
Definition: Input data from console to BASIC variables

The INPUT statement is a very versatile statement that is used to assign data from the console port to a variable. It is best be described by single feature explanations and examples.

1) Numeric variable assignment. A variable in the input list prompts with a '? ' and accepts characters up to a <carriage return>. This string is converted to binary and stored in the variable. If there is an error, the INPUT statement reprompts with '?? ' and attempts the assignment again.

```
LIST
 10  INPUT A
 20  PRINT A
 30  GOTO 10
RUN
? 1234
1234
? 12R<CR>?? _
```

2) String variable assignment. A string variable in the input list prompts with a ': ' and accepts up to 80 characters from the console terminal until a <carriage return>.

```
LIST
 10  DIM NAME[10]
 20  INPUT $NAME[0]
 30  PRINT "'";$NAME[0];"'"
 40  GOTO 20
RUN
: HOWDY PARTNER
'HOWDY PARTNER'
: _
```

3) Prompts. Any string constant found in the input list is echoed to the user console. If the input variable is preceded by a semicolon, the default prompt of '?' or ':' and space is suppressed. This enables the program to supply its own prompt.

```
LIST
 10  DIM D[10]
 20  INPUT "N=",N
 30  INPUT "N=";N
 40  INPUT "DATE=MNDYYR<8><8><8><8><8><8>";$D[0]
RUN
N=? 20
N=30
DATE=MNDYYR
     ^
```

4) Input maximum. The # operator sets the maximum number of characters that can be entered on any one variable assignment. Once the maximum has been set, it applies throughout the remainder of the input list unless another value is specified.

```
LIST
 200  INPUT #1;"DONE?";$I
 210  IF $I<>"Y": STOP
RUN
DONE?Y
STOP AT 210
```

(10.45 INPUT continued)

5) Input exact. The % operator sets an exact number of characters that must be entered on any one variable assignment. A <carriage return> is ignored until the exact number of characters has been entered.

```
LIST
 10  PRINT "MN/DY/YR<D>";
 20  INPUT %2;MN;"/";DY;"/";YR
RUN
MN/DY/YR
^
```

6) Error trapping. The ? operator in the input list, is used to specify a line number to which control transfers via a GOSUB statement if non-numeric data is entered where numeric data is required or control characters are entered for string inputs. The offending character value is found in SYS[0], with the line number in SYS[2].

```
LIST
 10  INPUT ?1000;"ENTER N  ";N
 20  GOTO 10
 1000  PRINT: PRINT "ERROR=";SYS[0];" AT";SYS[2]
 1010  POP: RETURN
RUN
ENTER N  <^C>
ERROR= 3 AT 10
ENTER N  _
```

7) Cursor addressing. When the @ operator is followed by two expressions, separated by a comma and enclosed in parentheses or brackets, each expression is evaluated and used to position the cursor at the respective X and Y locations.

```
LIST
 10  INPUT @[10,15];N
```

8) Screen control. If the @ operator is followed by a string, then certain letters specify control functions. Any letter may be preceded by a number which repeats the code that many times. These control letters are altered with the BFIX utility but are initially defined as follows:

```
LIST
 10  INPUT @"H10D15R";N
```

LETTER	VALUE	DEFINITION
C	<esc>*	CLEAR SCREEN
U	>0B	UP CURSOR
D	>0A	DOWN CURSOR
R	>0C	RIGHT CURSOR
L	>08	LEFT CURSOR
B	>0D	BEGINNING OF LINE
H	>1E	HOME CURSOR
S	<esc>Y	CLEAR TO END OF SCREEN
E	<esc>T	CLEAR TO END OF LINE
W	<esc>'	RESET WRITE PROTECT
P	<esc>&	SET WRITE PROTECT
(	<esc>)	START WRITE PROTECT
)	<esc>(	END WRITE PROTECT
Z	<esc>+	CLEAR UNPROTECTED
N	>09	SKIP TO NEXT FIELD

10.46 Function: INT

Format: INT <exp>
Definition: Returns greatest integer (floor) of <exp>

The INT function returns the greatest integer less than or equal to <exp>. For positive numbers, the functions INP and INT are identical, with the exception that INT has no limit on its range. Negative numbers return the next integer negative number if any fraction is found.

The INT function always rounds DOWN to the next lowest WHOLE number. It makes a positive number less positive, and makes a negative number more negative.

```
LIST
 10  INPUT "N=";N;
 20  PRINT ,INP[N],INT[N]
 30  GOTO 10
RUN
N=2.5            2              2
N=-2.5           -2             -3
N=-10            -10            -10
N=-32000.123     -32000         -32001
N=_
```

10.47 Function: KEY

Format: KEY <exp>
Definition: Returns last key value from port <exp>

The KEY function returns the value of the last key entered from a terminal. If <exp> is zero, then the user console port is sampled. If <exp> is nonzero, then it specifies which port to sample.

The value returned reflects the decimal value of the 7-bit character. If no key has been entered, the function returns a zero. If a key has been entered, it is removed from the input buffer and its value returned to the BASIC program.

```
LIST
 10  I=KEY[0]: IF I=0: GOTO 10
 20  $I=%I%0: PRINT $I;
 30  GOTO 10
RUN
ABCDEFGHI_
```

10.48 Statement: LABEL

Format: LABEL <sim-var>
Definition: Define line number label

The LABEL statement equates a simple variable to the LABEL statement line number. Thereafter, GOTOs and GOSUBs can reference the line by name rather than just by number.

The label variables are defined by the RUN and CLEAR statements. During a LIST function, LABEL statements are preceded by a blank line.

```
LIST
 100  GOSUB MENU
 110  PRINT MENU
 120  STOP

 500  LABEL MENU
 510  PRINT "I'M HERE!"
 520  RETURN
RUN
I'M HERE!
 500

STOP AT 120
```

10.49 Operator: LAND

Format: <exp1> LAND <exp2>
Definition: Logically 'AND' operands <exp1> and <exp2>

The LAND operator returns the logical 'AND' of operands <exp1> and <exp2>. The range of these arguments is plus or minus 65535. The result is returned in integer format.

```
;1 LAND 201; 1
;0FFH LAND 2000; 208
```

10.50 Function: LEN

Format: LEN[<string>]
Definition: Returns length of <string>

The LEN function returns the number of non-null characters in a string. It begins with the first character and counts until a null is encountered.

```
LIST
 10  DIM A[10]
 20  INPUT "STRING=";$A[0];
 30  PRINT "  LENGTH=";LEN[$A[0]]
 40  GOTO 20
RUN
ABCDEFG  LENGTH=7
<CR>  LENGTH=0
1234567890  LENGTH=10

_
```

10.51 Statement: LET

```
    Format:    LET <var> = <exp>
Definition:    Variable assignment
```

The LET statement is the basic assignment instruction of the BASIC language. The word LET is optional. Even though it might be entered, it does not LIST with the line. The LET statement has 12 different forms. In each example, the $A[0] array is first assumed to contain the 26 letters of the alphabet.

```
10 LET I=10
LIST
 10  I=10
DIM A[10]
$A[0]="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

1) Numeric assignment. The expression on the right of the equal sign is evaluated and stored in the variable on the left of the equal sign. This also applies to the returning of a function value by assigning it to the function name without arguments.

```
<var> = <exp>

PI=4*ATN 1
;PI; 3.14159265
```

2) String assignment. The string on the right of the equal sign is stored in the string variable on the left of the equal sign. Hex characters in angle brackets are not expanded. The assignment is terminated by a null character. If the string variable does not have enough storage reserved to handle the assignment, subsequent variables are overwritten. A string holds six times the variable size minus one. Thus, a simple variable holds only five characters. An array of ten elements holds 59 characters (10 x 6 - 1).

```
<string-var> = <string>

$A[0]="ABCDEFGHIJKL"
$I="YES"
;$A[0];$I;ABCDEFGHIJKLYES
```

3) Character pick. The assignment can be limited by following the string on the right of the equal sign with a comma and expression. The expression specifies the number of characters to be assigned to the variable. After the assignment is complete, an additional null character is stored to terminate the string. This assignment ignores all characters, including any nulls, in the source string.

```
<string-var> = <string> , <exp>

$A[0]="ABCDEFGHIJKLMNOP",5
;$A[0];ABCDE
```

(10.51 LET continued)

4) Replace characters. Characters are replaced within a string by following the string on the right of the equal sign with a semicolon and an expression. The expression specifies how many characters are to be replaced in the string variable on the left of the equal sign. No null character is written when the transfer is complete.

```
<string-var> = <string> ; <exp>

$A[0;5]="....";4
;$A[0];ABCD....IJKL
```

5) Concatenate strings. Strings are concatenated by means of the "&" operator. Strings on the right of the equal sign which are joined by the "&" operator are assigned to the string variable on the left of the equal sign. BASIC checks that the source byte is never equal to a previous destination byte, which would result in a CHOO CHOO effect. Such a condition terminates the assignment.

```
<string-var> = <string> & <string> ....

$A[0]="ABC"&"DEF"
$A[0]=$A[0]&"..."&"JKL"
;$A[0];ABCDEF...JKL
```

6) Delete characters. Characters are deleted from a string variable by following the equal sign with a back slash and an expression. The expression specifies how many characters are to be deleted. If the expression is zero or negative, no characters are deleted. The delete command deletes <exp> characters, or until a null character is found.

```
<string-var> = \ <exp>

$A[0;5]=\4
;$A[0];ABCDIJKLMNOPQRSTUVWXYZ
```

7) Insert characters. Characters are inserted into a string by following the equal sign with a backslash and a string. If the <string> is null, nothing is inserted.

```
<string-var> = \ <string>

$A[0;2]=\"...."
;$A[0];A....BCDEFGHIJKLMNOPQRSTUVWXYZ
```

8) Convert number to ASCII. An expression is converted to a string simply by assigning it to a string variable. The conversion is format free and uses the current number of digits in SYS[3]. The string is terminated by a null character.

```
<string-var> = <exp>

$A[0]=4*ATN 1
;$A[0]; 3.14159265
```

9) Convert number to ASCII with format. An expression is converted to a string using a print format by following the equal sign with a pound sign, then a string, followed by a comma and expression. The format string is the same as used by the PRINT statement. (See 10.74 PRINT.)

```
<string-var> = # <string> , <exp>

$A[0]=#"1-000-000-0000",8013752434
;$A[0];1-801-375-2434
```

(10.51 LET continued)

10) Convert number to HEX. An expression is converted to an ASCII string of four hex characters by following the equal sign with a pound sign and expression. The expression must be in the range of -32767 to 32767. A total of five characters are stored: four hex characters followed by a null.

```
<string-var> = # <exp>

$A[0]=#-2
;$A[0];FFFE
```

11) Convert byte. Individual bytes may be inserted into a string by following the equal sign with a percent sign and expression. The expression should range between 0 and 255 (8-bits). Many of these characters may be chained together by adding additional percent signs and expressions.

```
<string-var> = % <exp>

$A[0;2]=%65
;$A[0];AACDEFGHIJKLMNOPQRSTUVWXYZ
$A[0]=%65%66%67%0
;$A[0];ABC
```

12) Convert byte string. A string of hexadecimal characters is inserted by following the equal sign with a percent sign and a string. The only non-hexadecimal character allowed is a blank. Characters must consist of two hexadecimal characters.

```
<string-var> = % <string>

$A[0;2]=%"44 43 42 41 00"
;$A[0];ADCBAF
```

13) Convert ASCII to binary number. An ASCII string is converted to a binary number by assigning a string to a numeric variable. Since the conversion may have an error, the string is optionally followed by a comma and a variable to hold the delimiter character. The terminating byte is stored in the first byte of the variable. Hence, if the variable equals the null string, the conversion was successful. In any case, as many digits are converted and stored as possible.

```
<var> = <string> {,<var>}

$A[0]=4*ATN 1
N=$A[0],E
;N; 3.14159265
;"'";$E;"'";''
```

It is possible to chain many of the string assignments together in one assignment. Those operators allowing such chaining are %, \, #, and &.

```
$A[0]="-"%3EH#-2#3CH&"-"
;$A[0];->FFFE<-
$A[0]=#-9473%59H%32&"DUCK"
;$A[0];DAFFY DUCK
```

10.52 Command: LIST and LISTRP

```
   Format:     {<line #>} LIST
               {<line #>} LISTRP
Definition:    List user program to console
```

The LIST command outputs to the user console the current program in memory, in infix order. This is how a program is normally listed. Remember, a program may not list exactly the same way it is entered, since the program is changed to internal pseudo source tokens which are stored in Reverse Polish order. The LIST routine must then reconstruct an infix representation of these tokens, inserting parentheses where necessary to preserve operator precedence. Subscript parentheses list as brackets, while precedence parentheses list as parentheses.

The LISTRP command outputs to the user console the current program in memory in true token storage order. Each token is separated by a blank. Special characters are generated to show dimension operators. These are represented by a lower case 'd' followed by the number of dimensions. The LISTRP statement shows the exact order of program execution.

An optional <line #> can precede the LIST or LISTRP command to select where in the program the listing is to begin. There need not be a statement at <line #>, in which case the listing begins with the next greater line number.

The listing is temporarily interrupted by striking any key except <esc>. Striking any key again resumes the listing. The listing is terminated at any time by the <esc> key.

For every FOR statement, the next statement is indented by one blank. Every NEXT statement decrements the indentation by one. The ELSE and THEN statements add two blanks of indentation.

```
10 I=A/B+C
20 I=A/(B+C)
LIST
 10  I=A/B+C
 20  I=A/(B+C)
LISTRP
 10  I = A B / C +
 20  I = A B C + /
10 A(1,I*10)=B(C(1,2),SQR(V))
20 X=(A*B)*(C*D)
LIST
 10  A[1,I*10]=B[C[1,2],SQR[V]]
 20  X=A*B*(C*D)
LISTRP
 10  1 I 10 * d2 A = 1 2 d2 C V d1 SQR d2 B
 20  X = A B * C D * *
```

10.53 Operator: LNOT

Format: LNOT <exp>

Definition: Logically complement operand <exp>

```
;LNOT 0FEH; -255
;LNOT 1; -2
```

The LNOT operator returns the logical 1's complement of <exp1>. The range of the argument is plus or minus 65535. The result is returned in integer format.

10.54 Statement: LOAD

Format: LOAD <string>

Definition: Load program from PDOS file

The LOAD statement loads an ASCII text BASIC program into the current workspace. If a program already exists, the new program is merged and overlayed where conflicting line numbers are found. The LOAD command opens the file using a read only open, and closes the file when the EOF is found.

If an error occurs during the load, the offending line is printed and the load continues. The LOAD statement can appear in a program and be used with PURGE for runtime overlays.

```
LOAD "PROGRM1"
*READY
LIST
  10  PRINT "PROGRAM 1"
  20  LOAD "PROGRM2"
  30  GOTO 10
LOAD "PROGRM2"
*READY
LIST
 10  PRINT "PROGRAM 2"
 20  LOAD "PROGRM1"
 30  GOTO 10
RUN
PROGRAM 2
PROGRAM 1
PROGRAM 2
....
```

10.55 Statement: LOCAL

Format: LOCAL <sim-var> {,...}

Definition: Declare a simple variable local to a function

The LOCAL statement is used within a function definition to add simple variables to the local dummy variable list. Simple variables declared to be local are different from variables of the same name outside the body of the function.

Local variables are redefined each time the function call is made and dropped when the function is exited. They are stacked during recursion and not affected while other functions are called from within a function.

```
LIST
 100  INPUT "C1 = (";R1;",";I1;")"
 110  INPUT "C2 = (";R2;",";I2;")"
 120  CALL FNCDIV[R1,I1,R2,I2,X,Y]
 130  PRINT "C1/C2=(";X;",";Y;")"
 140  GOTO 100
 500  DEFN FNCDIV[R1,I1,R2,I2,R,I]
 510  LOCAL K1,K2
 520  IF I2=R2: IF I2=0: R=1E75: I=1E75: FNEND
 530  IF ABS R2<ABS I2
 540    THEN K1=R2/I2: K2=I2+K1*R2
 550    THEN R=(R1*K1+I1)/K2: I=(I1*K1-R1)/K2
 560    ELSE K1=I2/R2: K2=R2+K1*I2
 570    ELSE R=(R1+I1*K1)/K2: I=(I1-R1*K1)/K2
 580  FNEND
RUN
C1 = (1,0)
C2 = (0,1)
C1/C2=( 0, -1)
C1 = (2,2)
C2 = (4,-1)
C1/C2=( 0.352941176, 0.588235294)
C1 = (_
```

10.56 Function: LOG

Format: LOG <exp>
Definition: Returns natural log of <exp>

The LOG function returns the natural logarithm of <exp>. The expression must be positive. The base 10 log is obtained by multiplying the natural logarithm by 0.434295.

The LOG function is the inverse of the EXP function. Hence, EXP of the LOG of N returns N.

```
LIST
 10  FOR I=0.5 TO 2 STEP 0.1
 20   PRINT LOG I; TAB`LOG[I]*20+30;"*"
 30  NEXT I
RUN
 -0.693147181   *
 -0.510825624      *
 -0.356674944         *
 -0.223143551            *
 -0.105360516              *
 0                            *
 0.0953101798                  *
 0.182321557                     *
 0.262364264                       *
 0.336472237                        *
 0.4054651                            *
 0.470003629                           *
 0.530628251                            *
 0.587786665                             *
 0.641853886                              *
 0.693147181                               *

STOP AT 30

;EXP(LOG(2)); 2
```

10.57 Operator: LOR

Format: <exp1> LOR <exp2>
Definition: Logically 'OR' operands <exp1> and <exp2>

The LOR operator returns the logical 'OR' of arguments <exp1> and <exp2>. The range of the arguments is plus or minus 65535. The result is returned in integer format.

```
;08000H LOR 10; -32758
;020H LOR 010H; 48
```

10.58 Operator: LXOR

Format: <exp1> LXOR <exp2>
Definition: Exclusive 'OR' operands <exp1> and <exp2>

The LXOR operator returns the logical exclusive 'OR' of arguments <exp1> and <exp2>. The range of the arguments is plus or minus 65535. The result is returned in integer format.

```
;022H LOR 042H; 98
;022H LXOR 042H; 96
```

10.59 Variable: MAIL

Format: MAIL[<exp>]
Definition: Global array for intertask communication

The MAIL variable is a global array that can be referenced by any other BASIC program or assembly language task. The array is a single dimensioned array of ten elements (MAIL[0] through MAIL[9]). MAIL[0] is located at address >2200 to >223F of the PDOS system RAM.

```
LIST
 100  MAIL[1]=N  !PASS MOVE
 110  MAIL[0]=1  !SIGNAL MOVE READY
 120  IF MAIL[0]=1: GOTO 120  !WAIT
....
```

10.60 Function: MCH

Format: MCH[<string1>,<string2>]
Definition: Returns number of matching characters

The MCH function returns the number of characters in which <string1> and <string2> agree. <string1> can have a wild card character '*' which always matches.

```
LIST
 10  DIM A[10]
 20  INPUT $A[0]
 30  PRINT "ABCDEFG =";MCH["ABCDEFG",$A[0]]
 40  PRINT "AB**EF* =";MCH["AB**EF*",$A[0]]
 50  GOTO 20
```

10.61a Function: MEM

Format: MEM <exp>
Definition: Returns byte value of memory location <exp>

The MEM function returns the value of memory location <exp>. The result is an integer ranging from 0 to 255.

```
LIST
 10  FOR I=0 TO 99
 20   IF FRA[I/10]=0: PRINT
 30   $D[0]=#MEM[I]
 40   PRINT $D[0;3];" ";
 50  NEXT I
RUN

2F DC 00 CC 22 78 22 98 22 6C
22 8C 2F DC 03 98 2F 60 05 9C
2F 60 05 9C 2F 60 05 9C 22 60
22 80 22 54 22 74 22 48 22 68
22 3C 22 5C 22 30 22 50 22 24
22 44 22 18 22 38 22 0C 22 2C
22 00 22 20 2E 16 17 4A 2E 16
17 3E 2E 16 17 64 2E 16 17 56
2E 16 18 D8 2E 16 19 AA 2E 16
1A 80 2E 16 18 3E 2E 16 40 D6
STOP AT 50
```

10.61b Statement: MEM

Format: MEM[<exp1>]=<exp2>
Definition: Store byte in memory

The MEM statement stores the byte value of <exp2> in memory at location <exp1>. The range of <exp2> is from 0 to 255. A larger value stores only the right most 8 bytes.

```
MEM(02F65H)=0  !CLEAR SECONDS
```

10.62a Function: MEMW

Format: MEMW <exp>
Definition: Returns word value of memory location <exp>

The MEMW function returns the word value of memory location <exp>. The result is an integer ranging from -32767 TO 32768.

```
LIST
 10  FOR I=0 TO 99 STEP 2
 20   IF FRA[I/10]=0: PRINT
 30   $D=#MEMW[I]
 40   PRINT $D;" ";
 50  NEXT I
RUN

2FDC 00CC 2278 2298 226C
228C 2FDC 0398 2F60 059C
2F60 059C 2F60 059C 2260
2280 2254 2274 2248 2268
223C 225C 2230 2250 2224
2244 2218 2238 220C 222C
2200 2220 2E16 174A 2E16
173E 2E16 1764 2E16 1756
2E16 18D8 2E16 19AA 2E16
1A80 2E16 183E 2E16 4DD6
STOP AT 50
```

10.62b Statement: MEMW

Format: MEMW[<exp1>]=<exp2>
Definition: Store 16 bit word in memory

```
MEMW[0090H]=09BAAH  !SET CLEAR SCREEN
```

The MEMW statement stores the 16-bit integer value of <exp2> in memory at location <exp1>. The range of <exp2> is from -32767 to 32767.

10.63a Function: MEMP

Format: MEMP[<exp1>,<exp2>]

Definition: Returns a 6-byte BASIC number from address <exp1>, page <exp2>

The MEMP function returns a 6-byte BASIC number from address <exp1>, page <exp2>. On pages systems, such as TM990/101MA, <exp1> is an absolute address and <exp2> selects a extended memory page. <exp1> ranges from >0000 to >FFFF and <exp2> ranges from 0 to 7.

On memory mapped systems, such as TM990/102, <exp1> is a page displacement and ranges from >0000 to >0FFF. <exp2> selects a mapped page and ranges from 0 to 64.

```
LIST
 10  IF MAIL[0]=0: SKIP -1  !WAIT FOR ADDRESS
 20  FOR I=0 TO 10
 30   PRINT MEMP[MAIL[0],1];
 40  NEXT I
```

10.63b Statement: MEMP

Format: MEMP[<exp1>,<exp2>]=<exp3>

Definition: Store 6-byte BASIC number <exp3> at address <exp1>, page <exp2>

The MEMP stores a 6-byte BASIC number <exp3> at address <exp1>, page <exp2>. On pages systems, such as TM990/101MA, <exp1> is an absolute address and <exp2> selects a extended memory page. <exp1> ranges from >0000 to >FFFF and <exp2> ranges from 0 to 7.

On memory mapped systems, such as TM990/102, <exp1> is a page displacement and ranges from >0000 to >0FFF. <exp2> selects a mapped page and ranges from 0 to 64.

```
MEMP[0,55]=4*ATN 1
```

10.64 Function: NCH

Format: NCH[<string>]
Definition: Returns numeric value of 1st character of <string>

The NCH function returns the numeric value of a character. The first byte of <string> is returned as an integer and ranges from 0 to 255.

```
LIST
 10  INPUT #1;$A;
 20  PRINT TAB 10;"VALUE=";NCH[$A]
 30  GOTO 10
RUN
:A        VALUE= 65
:*        VALUE= 42
:
```

10.65 Command: NEW

Format: NEW
Definition: Clear user program and variable space

The NEW command clears the user memory of all program code in preparation for entering or loading a new program. In addition, all buffers and stacks are reset, and the following initialized:

AINC = 10	Auto increment size
COMZ = 10	COM[] array size
DGTS = 8	Format free number size
EXTZ = 20	EXTERNAL table size
FNSS = 8	FOR/NEXT stack size
GSSS = 20	GOSUB stack size
UNIT = 1	Output unit

```
LIST
 10  REM PROGRAM #1
 20  I=SIN[1]*SIN[1]+COS[1]*COS[1]
SIZE
PRGM:46
VNAM:0
VARS:2
FREE:31614
NEW
*READY
SIZE
PRGM:0
VNAM:0
VARS:0
FREE:31662
```

10.66 Statement: NEXT

Format: NEXT <sim-var>
Definition: Foot of a BASIC loop

The NEXT statement marks the end of a FOR loop. The argument must be a simple variable and match the variable name used in the corresponding FOR statement. The NEXT statement adds the STEP value to the variable, updates it in memory, and then checks to see if the loop has been completed. If the condition has not been met, execution continues immediately after the FOR statement (which may be on the same line). If the condition is met, execution continues with the next statement after the NEXT.

For a pre-test to work, the NEXT statement must be the first word on a program line.

During a LIST or LISTRP command, each NEXT statement decrements the line indentation by one.

```
LIST
 10 FOR I=1 TO 3
 20  FOR J=1 TO 2
 30   PRINT I,J
 40  NEXT J
 50 NEXT I
RUN
 1               1
 1               2
 2               1
 2               2
 3               1
 3               2

STOP AT 50
```

10.67 Statement: NOESC

Format: NOESC
Definition: Disable ESC key for break function

The NOESC statement disables the <esc> key for breaking program execution. The <esc> key is again allowed when an ESCAPE statement is executed or the program returns to keyboard mode.

NOESC has no effect in keyboard mode. Since at least one statement is executed after a RUN, a NOESC statement is guaranteed to be executed, thus protecting any program from operator breaks.

```
LIST
 10 NOESC
 20 FOR I=1 TO 1920
 30  PRINT "N";
 40 NEXT I
 50 ESCAPE
 60 FOR I=1 TO 1920
 70  PRINT "Y";
 80 NEXT I
RUN
```

(The program fills the screen with N's without operator interruption. Only after the Y's appear can the operator break execution with the <esc> key.)

10.68 Operator: NOT

Format: NOT <exp>
Definition: Returns TRUE if <exp>=0, else FALSE

The Boolean operator NOT returns TRUE (1) when the expression is zero and FALSE (0) when the expression is nonzero.

```
LIST
 10  A=6: B=0: C=0
 20  IF NOT A+B-0: PRINT "NO"
 30   ELSE PRINT "YES"
 40  IF NOT C: PRINT "CORRECT"
 50  PRINT NOT C
RUN
YES
CORRECT
 1

STOP AT 60
```

10.69 Statement: ON

Format: ON <exp> : GOTO <line #>,...
ON <exp> : GOSUB <line #>,...
ON <exp> : <var> = <exp1>,<exp2>,...

Definition: Case statement for GOTO, GOSUB, and LET

The ON statement selects a line number for a GOTO or GOSUB from a list of line numbers separated by commas. Or, a variable is assigned a value from a list of expressions separated by commas. The expression <exp> is integerized and the value used to select the appropriate parameter.

If the expression is out of range (less than one or greater than the number of expressions in the list), the program continues with the next statement.

```
LIST
 10  INPUT I;
 20  ON I: J=4,3,2,1
 30  ON I: GOTO 100,200,300
 40  PRINT ,"LINE 30"
 50  GOTO 10
 100  PRINT ,"LINE 100",J: GOTO 10
 200  PRINT ,"LINE 200",J: GOTO 10
 300  PRINT ,"LINE 300",J: GOTO 10
RUN
? 1            LINE 100         4
? 3            LINE 300         2
? 0            LINE 30
? _
```

10.70 Statement: OPEN

Format: OPEN <string>,<var>
Definition: Open a PDOS file for sequential access

```
LIST
 10  OPEN "FILE/1",FILID
 20  BINARY 1,FILID,3,I,J,K
```

The OPEN statement opens a file for sequential access and returns the FILE ID in <var>. The file name, optional extension, and optional disk number are included in the string.

The FILE ID is used for all subsequent file references.

10.71 Operator: OR

Format: <exp1> OR <exp2>
Definition: Returns TRUE if <exp1> or <exp2> is nonzero

```
LIST
 10  A=1: B=2: C=0
 20  IF A<B OR C: PRINT "A<B"
 30  IF A>B OR C=1: PRINT "A>B OR C=1"
RUN
A<B

STOP AT 30
```

The Boolean operator OR evaluates TRUE (1) if either or both <exp1> and <exp2> evaluate nonzero. OR returns FALSE (0) only when both <exp1> and <exp1> are zero. Note: <exp1> and <exp2> cannot be strings.

10.72 Statement: PDOS

Format: PDOS <var>,{list}

Definition: Read parameter for PDOS command list

The PDOS statement is used to retrieve parameters from a PDOS command list. <var> is loaded with the number of parameters returned. The PDOS {list} consists of variables or string variables separated by commas.

If a variable is found, the parameter is evaluated, fixed, and stored in the variable. The range is from -32767 to 32767.

If a string variable is found, the complete parameter is returned as a string.

```
LIST
 10  PDOS N,I
 20  IF N=0: BYE
 30  PRINT I;" FACTORIAL =";FNFACT[I]
 40  GOTO 10
 100  DEFN FNFACT[N]
 110  IF N<=1: FNFACT=1: FNEND
 120  FNFACT=N*FNFACT[N-1]
 130  FNEND
DEFINE "FACT"
SAVE "FACT"
*READY
BYE
.FACT 2,4,6,8,10,20
 2 FACTORIAL = 2
 4 FACTORIAL = 24
 6 FACTORIAL = 720
 8 FACTORIAL = 40320
 10 FACTORIAL = 3268800
 20 FACTORIAL = 2.432902E18
._
```

10.73 Statement: POP

Format: POP

Definition: Remove an entry from the GOSUB stack

The POP statement removes the last GOSUB return address from the GOSUB stack.

A GOSUB, ERROR, or INPUT '?' operator places a return address on the GOSUB stack. The RETURN statement pops the top entry and uses it to continue execution after the call. The POP statement is similar to a RETURN except that it does not do a transfer.

POP is particularly useful when exiting from a subroutine to multiple places. It also is necessary when acknowledging errors.

```
LIST
 10  ERROR 1000
 20  DIM A[10]
 30  INPUT "NAME=";$A[0]
 40  OPEN $A[0],F
 50  STOP
 1000  POP
 1010  IF SYS[1]=50
 1020    THEN PRINT "INVALID FILE NAME": GOTO 10
 1030  STOP
RUN
NAME=&FILE
INVALID FILE NAME
NAME=_
```

10.74 Statement: PRINT

```
    Format:     PRINT <print list>
Definition:     Output data to user console
```

The PRINT statement outputs to the user console, in ASCII format, any string or expression found in the <print list>. Output is directed to the terminal or file, depending upon the UNIT and SPOOL instructions.

The PRINT statement is very versatile and is explained with examples. PRINT items must be separated by at least a semicolon delimiter. Other valid delimiters are TAB and comma.

If a semicolon is the first character of a statement, it is changed to a PRINT command.

1) Strings. A string constant or string variable is printed when found in the parameter list. The string is examined for any ASCII literals which are delimited by angle brackets (e.g. <0A>).

```
PRINT <string>

$I="NO"
;"THE ANSWER IS ";$I;THE ANSWER IS NO
```

2) Expressions. Any expression found in the print list is printed in format-free form (unless a format has been specified by the "#" operator). In format-free form, a space always precedes the number and if necessary, the output changes to scientific notation in order to accommodate numbers too small or too large.

```
PRINT <exp>

;4*ATN 1; 3.14159265
```

3) Suppress <carriage return>. A semi-colon at the end of a PRINT statement suppresses the <carriage return> and <line feed>. (A TAB or comma at the end of a PRINT statement does the same.)

```
PRINT <exp>;

;"HELLO";HELLO
;"HELLO"HELLO


_
```

4) Print zones. The print columns 16, 32, 48, 64, ... are defined as print zones. When a comma is found in the print list, spaces are output until the next zone is reached. If the comma is the last item of the PRINT statement, a <carriage return> is suppressed.

```
PRINT <exp>,

;1,2,3; 1       2               3
;"A","B","C";A  B               C
```

(10.74 PRINT continued)

5) TAB function. The TAB function evaluates and fixes <exp> and outputs spaces or back spaces until it agrees with the system column counter (>18E(9)). If the TAB is the last item of the PRINT statement, a <CR> is suppressed.

```
PRINT TAB <exp>

LIST
 10  FOR I=1 TO 10
 20   PRINT TAB I^2/4;"*"
 30  NEXT I
RUN
*
*
 *
   *
     *
       *
          *
             *
                 *
                     *

STOP AT 30
```

6) Cursor addressing. When the "@" operator is followed by two expressions, separated by a comma and enclosed in parentheses or brackets, each expression is evaluated and used to position the cursor at the respective X and Y position. Position @[0,0] is defined as the home position.

```
PRINT @[<exp1>,<exp2>]

LIST
 10  PRINT @"C"
 20  FOR I=1 TO 8
 30   FOR J=1 TO 2*I-1
 40    PRINT @[I,20-I+J];"*"
 50   NEXT J
 60  NEXT I
RUN
(screen clears)
                   *
                  ***
                 *****
                *******
               *********
              ***********
             *************
            ***************

STOP AT 50
```

7) Print hex number. If an expression is preceded by a pound sign, then the fixed number is printed as a four character hexadecimal number.

```
;100; 100
;#100;0064
;#-123;FF85
```

(10.74 PRINT continued)

8) Screen commands. If the "@" operator is followed by a string, then specific letters specify control function. Any letter may be preceded by a number which repeats the code that many times. These control letters are defined as follows:

LETTER	VALUE	DEFINITION
C	<esc>*	CLEAR SCREEN
U	>0B	UP CURSOR
D	>0A	DOWN CURSOR
R	>0C	RIGHT CURSOR
L	>08	LEFT CURSOR
B	>0D	BEGINNING OF LINE
H	>1E	HOME CURSOR
S	<esc>Y	CLEAR TO END OF SCREEN
E	<esc>T	CLEAR TO END OF LINE
W	<esc>'	RESET WRITE PROTECT
P	<esc>&	SET WRITE PROTECT
(	<esc>)	START WRITE PROTECT
)	<esc>(	END WRITE PROTECT
Z	<esc>+	CLEAR UNPROTECTED
N	>09	SKIP TO NEXT FIELD

```
;@"C2D5R";"HELLO"

        HELLO
```

9) PRINT formatting. Numeric output can be formatted to right justify, float a sign, dollar sign, or angle brackets, or insert commas or periods. The pound sign is followed by a string which specifies the format. This format applies throughout the rest of the PRINT statement unless reset or changed. Numbers are rounded on the last printed digit.

Format characters are defined as follows:

Character	Digit holder	No digit
9	Yes	Space
0	Yes	0
$	Yes	Floats $
S	Yes	Floats sign
<	Yes	Floats < on negative
>	No	> on negative
E	No	Print sign
.	Decimal point	
,	Prints only if preceded by digit	
^	Replaced with period	

A digit holder is defined as a position where a digit can be printed. A floater appears only once and to the left of the first digit. If there are not enough digit holders to handle the edited number, the format is replaced with asterisks. All non-formatting characters remain in the format mask and are printed.

```
PRINT #<string>



LIST
 10  DIM FORMAT[10]
 20  INPUT $FORMAT[0];
 30  PRINT #$FORMAT[0],543.34,-12345.67,-0.05
 40  GOTO 20
RUN
: 99999           543         12346
: 00000         00543         12346         00000
: 990           543           ***              0
: 000-00-0000  000-00-0543   000-01-2346   000-00-0000
: $$$$$.00       $543.34     12345.67          $.05
: $$$$0.00       $543.34     12345.67         $0.05
: <<<<<0.00>      543.34    <12345.67>       <0.05>
: 99999.00E       543.34     12345.67-          .05-
: 999,999.00E       543.34    12,345.67-          .05-
: 999,990.99E       543.34    12,345.67-         0.05-
: SSS,SS0.00        543.34   -12,345.67         -0.05
: 0^0^0^0^0^0  0.0.0.5.4.3   0.1.2.3.4.6   0.0.0.0.0.0
: _
```

10.75 Statement: PURGE

Format: PURGE <line #1> TO <line #2>
Definition: Delete program segment

The PURGE statement deletes program statements from <line #1> up to and including <line #2>. The given line numbers need not exist in the program.

PURGE is used in connection with the LOAD command to do program chaining and overlays.

```
LIST
 10  REM
 20  REM
 30  REM
 40  REM
 50  REM
 60  REM
PURGE 20 TO 45
LIST
 10  REM
 50  REM
 60  REM
```

10.76 Statement: READ

Format: READ <var>,...
Definition: Read program data from DATA statements

The READ statement reads data sequentially from the program DATA statements. Either numeric or string data can be read but the type must be the same for both the READ variable and the DATA value. READ variables are separated by commas.

```
LIST
 10  READ A,B,$C
 20  PRINT $C,B,A
 30  DATA 456,23
 40  DATA "HELLO"
RUN
HELLO           23              456

STOP AT 40
```

A Data List Pointer is maintained by the system and indicates where the next data item is located. This pointer is set to the first data item when the program is RUN and thereafter adjusted by a READ or RESTORE statement.

The READ statement translates ASCII literals within strings to their one byte equivalent. An ASCII literal is a hex number enclosed in angle brackets.

```
  990  DATA "HR:MN:SC<D>"
```

10.77 Statement: REM

Format: REM <characters>
Definition: Program remark for documentation

The REMark statement is used to enter ASCII documentation in a program. Once a REM statement is encountered, BASIC ignores the rest of the program line and moves to the next line.

Remarks are added at the end of any line by using an exclamation point followed by a string of characters.

```
LIST
 10  REM PROGRAM BEGINNING
 20  PRINT 4*ATN 1  !PRINT PI
RUN
 3.14159265

STOP AT 20
```

10.78 Statement: RENAME

Format: RENAME <string1> TO <string2>
Definition: Rename a PDOS file

The RENAME statement renames the file specified by <string1> to <string2>. This command alters the file directory level of the file in <string1> by specifying the new directory level in <string2>.

```
RENAME "OLDFILE" TO "NEWFILE"
RENAME "FILE" TO "255"
```

10.79 Statement: RESET

Format: RESET {<exp>}
Definition: Close all PDOS files by task or disk

```
RESET 0          (Disk 0 could be safely removed)
RESET            (All current task files are closed)
```

The RESET statement closes all open files either by task or by disk number. If no expression follows the RESET statement, then all files associated with the current user task are closed. If an expression is given, then it is evaluated and all files open on that disk number are closed.

In either case, the SPOOL UNIT and assigned input FILE ID's are cleared.

10.80 Statement: RESTORE

Format: RESTORE {<exp>}
Definition: Set program DATA pointer

The RESTORE statement is used to specify where the next DATA item is located. Normally, the data pointer moves sequentially through the program as items are READ. However, one may wish to re-read many items, or even have random access into a DATA list.

The RESTORE statement has the following modes of operation:

1) If RESTORE has no argument, or the argument evaluates to zero, then the data pointer is set to the first DATA item in the program.

2) If RESTORE has a positive argument, then the data pointer is moved forward in the program <exp> items following the RESTORE statement.

3) If RESTORE has a negative argument, then the data pointer is moved forward in the program <exp> items from the beginning of the program.

All DATA statements form a large pool of data, regardless of where they are located in a program. The RESTORE statement randomly accesses any DATA item.

```
LIST
 10  DATA 1,2,3
 20  GOSUB 1000
 30  DATA 4,5,6
 40  RESTORE
 50  GOSUB 1000
 60  RESTORE -7
 70  DATA 7,8,9
 80  GOSUB 1000
 90  RESTORE 3
 100  DATA 10,11,12
 110  DATA 13,14,15
 120  GOSUB 1000
 130  STOP
 1000  FOR I=1 TO 4
 1010   READ X: PRINT X;
 1020  NEXT I
 1030  PRINT
 1040  RETURN
RUN
 1 2 3 4
 1 2 3 4
 7 8 9 10
 12 13 14 15

STOP AT 130
```

10.81 Statement: RETURN

Format: RETURN {<exp>}
Definition: Pop entry from GOSUB stack and return

The RETURN statement is used to exit a subroutine. Any GOSUB operation places a return address on the GOSUB stack. The RETURN pops the top item from the stack into the program counter, and thus continues execution immediately after the call.

The RETURN has an optional parameter which is used to adjust its return address.

If the expression is zero, then the RETURN goes immediately to the next line following the call and not execute any further statements on the same line as the GOSUB.

If the expression is nonzero, then a RETURN: SKIP <exp> is executed. Thus a RETURN -1 repeats the call.

```
LIST
 10  DIM N[10]
 20  GOSUB 100: GOSUB 200
 30  STOP
 100  INPUT "NAME=";$N[0]
 110  IF $N[0]="": RETURN 0
 120  RETURN
 200  PRINT TAB 10;$N[0]
 210  RETURN -1  !REPEAT AGAIN
RUN
NAME=TOM
          TOM
NAME=JOHN
          JOHN
NAME=<CR>

STOP AT 30
```

10.82 Variable: RND

Format: RND
Definition: Variable with random value between 0 and 1

The RND variable returns a random number between 0 and 1 every time it is accessed. The random numbers are generated from a seed. This seed is altered with the SYS[13] variable.

Each new seed is generated by the following linear congruential sequence:

```
        X[n+1] = ( X[n] * A + 13849 ) mod 2^16
```

```
LIST
 10  DEFN FNRND[X]=INT[X*RND]
 20  FOR I=1 TO 10
 30   PRINT I,FNRND[I]
 40  NEXT I
RUN
 1              0
 2              1
 3              1
 4              2
 5              1
 6              5
 7              2
 8              4
 9              2
 10             6

STOP AT 40
```

10.83 Statement: ROPEN

Format: ROPEN <string>,<var>

Definition: Open for Random access a PDOS file

The ROPEN statement opens a file specified by <string>, for random access. The FILE ID is returned in <var>. All subsequent access to the file is through the FILE ID.

The END-OF-FILE marker on a random file is changed only when the file has been extended. All data transfers are buffered through a channel buffer and data movement to and from the disk is by full sectors.

(FDATA has BASIC numbers 0 to 100 with squares and cubes.)

```
LIST
 10  ROPEN "FDATA",FILID
 20  I=INT[100*RND]
 30  BINARY 1,FILID;4,I,18,0;3,J,K,L
 40  PRINT I,J;K;L
 50  GOTO 20
RUN
 62             62 3844 238328
 1              1 1 1
 15             15 225 3375
 14             14 196 2744
 48             48 2304 110592
 41             41 1681 68921
 35             35 1225 42875
 19             19 361 6859
 74             74 5476 405224
 8              8 64 512
 90             90 8100 729000
 93             93 8649 804357
```

10.84 Statement: RUN

Format: RUN
RUN <string>

Definition: Begin program execution

The RUN statement enters run mode and begins program execution at the statement with the smallest line number. All variables are cleared and all system stacks and pointers are reset.

If the RUN statement has a string argument, BASIC chains to the specified file. This file need not be a BASIC program, as chaining to assembly language programs is allowed. A NEW command is executed before the new program is loaded.

```
LIST
 10  INPUT "SELECT PROGRAM, N=",N
 20  ON N: GOTO 100,200,300
 30  PRINT "TRY AGAIN!"
 40  GOTO 10
 100  RUN "PROGRM1"
 200  RUN "PROGRM2"
 300  RUN "PROGRM3"
RUN
SELECT PROGRAM, N=5
TRY AGAIN!
SELECT PROGRAM, N=1
```

(PROGRM1 is loaded and executed)

10.85 Command: SAVE

```
    Format:     SAVE <string>
Definition:     Save user program in PDOS file
```

```
SAVE "PRGM"
*READY
```

The SAVE command saves the current program into the PDOS file specified by <string>, in ASCII text format. This is equivalent to a LIST to the file. The program is stored as ASCII characters in infix order.

A SAVEd program is given the PDOS type 'EX' and is executed again from PDOS simply by entering the file name. The program format is compatible with the LOAD statement.

10.86 Command: SAVEB

```
    Format:     SAVEB <string>
Definition:     Save user program as tokens in PDOS file
```

```
SAVEB "PROGRM"
*READY
```

The SAVEB command saves the current program into the PDOS file specified by <string>. The format, however, is in untranslated binary pseudo source tokens. The file is typed 'BX' and can only be run by RUN or PDOS. It cannot be LOADed.

This format has many advantages. First, it requires less disk storage than the ASCII format. Second, the load time is dramatically reduced! Third, the file is compatible with the standalone BASIC interpreter run module and is burned directly into EPROM's.

10.87 Function: SGN

Format: SGN <exp>
Definition: Returns signed value of <exp>: -1=negative, 0=zero, 1=positive

The SGN function returns a one, negative one, or zero, depending on the sign of the argument. If the expression evaluates to a positive number, a one is returned. If the expression is zero, a zero is returned. Finally, if the expression is negative, a negative one is returned.

```
LIST
 10  INPUT "N=";N;
 20  PRINT  TAB 10;"SGN[N]=";SGN[N]
 30  GOTO 10
RUN
N=12      SGN[N]= 1
N=0       SGN[N]= 0
N=-300    SGN[N]= -1
N=
```

10.88 Function: SIN

Format: SIN <exp>
Definition: Returns sine value of radian <exp>

The SIN function returns the sine of the expression. <exp> is given in radians. To obtain degrees, multiply the expression by 0.01745329.

```
LIST
 10  FOR I=0 TO 7 STEP ATN 1/2
 20   PRINT SIN I; TAB SIN[I]*14+30;"*"
 30  NEXT I
 40  STOP
RUN
 0                            *
 0.38268343                        *
 0.70710678                            *
 0.92387953                               *
 1                                          *
 0.92387953                               *
 0.70710678                            *
 0.38268343                        *
 0                            *
 -0.38268343            *
 -0.70710678        *
 -0.92387953     *
 -1             *
 -0.92387953     *
 -0.70710678        *
 -0.38268343            *
 -2.2858095E-11              *
 0.38268343                        *

STOP AT 40
```

10.89 Command: SIZE

Format: SIZE
Definition: List user program size and available memory

The SIZE command lists to the user console size parameters pertaining to BASIC memory usage. These parameters are defined as follows:

PRGM: Program size. This value is the sum of the program tokens and statement line number table in bytes. It also represents the size of the EPROM module. (Add 12 to this value for initialization parameters when burning EPROMs.)

VNAM: Variable names. This value is the number of bytes required to store the variable names.

VARS: Variable storage. This value is the sum of the variable name table, variable definition table, and variable storage in bytes. (This number is used in the approximation of the RAM requirements for a run module.)

FREE: Available storage. All available storage is listed as FREE memory in bytes. This memory must be shared by new variable names and definitions, program lines, and variable storage.

The SYS function monitors the memory partition pointers and can adjust the size of the GOSUB, FOR/NEXT, and EXTERNAL tables. The SYS values are defined as follows:

```
SYS[20] = BUS = Beginning of User Storage
SYS[21] = SLT = Statement Line Table
SYS[22] = VNT = Variable Name Table
SYS[23] = VDT = Variable Definition Table
SYS[24] = NVD = Next Variable Definition
SYS[25] = NVS = Next Variable Storage
SYS[26] = GSS = GOSUB Stack
SYS[27] = FNS = FOR/NEXT Stack
SYS[28] = EUS = End User Storage
SYS[35] = EXT = EXTERNAL Table
SYS[29] = EUM = End User Memory
```

PRGM= A+B	Program size
VNAM= C	Variable names
VARS= D+G	Variable Size
FREE= E+F	Available storage

```
                ¦       ¦
BUS             ¦-------¦
                ¦   A   ¦ Program storage
SLT             ¦-------¦
                ¦   B   ¦ Statement numbers
VNT             ¦-------¦
                ¦   C   ¦ Variable names
VDT             ¦-------¦
                ¦   D   ¦ Variable definitions
NVD             ¦-------¦ End of definitions
                ¦   E   ¦
   R10=>        ¦       ¦ Heap pointer
                ¦       ¦
                ¦   F   ¦ FREE space
NVS             ¦-------¦
                ¦   G   ¦ Variable storage
GSS             ¦-------¦
                ¦ 20x4b ¦ GOSUB stack
FNS             ¦-------¦
                ¦ 8x18b ¦ FOR/NEXT stack
EXT             ¦-------¦
                ¦ 20x2b ¦ EXTERNAL table
EUS             ¦_______¦ End User Storage
EUM
```

```
SIZE
PRGM:0
VNAM:0
VARS:0
FREE:31662
N=6
DIM A(10)
SIZE
PRGM:0
VNAM:2
VARS:80
FREE:31580
10 X=100
SIZE
PRGM:10
VNAM:2
VARS:82
FREE:31568
```

10.90 Statement: SKIP

Format: SKIP <exp>
Definition: Conditional program transfer

The SKIP statement causes program execution to skip the number of program lines specified by <exp>.

If the expression is zero, execution continues on the next line. A value of -1 would execute the current line again.

The SKIP statement is a very fast transfer, but caution must be used. Since these transfers do not appear in TRACE 2, they cannot be interrupted with the ESCAPE key, and cause problems when statements are added or deleted.

```
LIST
 10  REM MAKE NONSENSE
 20  SKIP INT[5*RND]
 30  PRINT "EATS ";: GOTO 10
 40  PRINT "THE CAT ";: GOTO 10
 50  PRINT "LICKS ";: GOTO 10
 60  PRINT "TODAY.": GOTO 10
 70  PRINT "THE DOG ";: GOTO 10
RUN
LICKS TODAY.
THE CAT EATS LICKS TODAY.
THE DOG TODAY.
THE CAT TODAY.
EATS TODAY.
THE CAT LICKS THE CAT THE CAT EATS THE DOG TODAY.
EATS THE DOG TODAY.
```

10.91 Statement: SOPEN

Format: SOPEN <string>,<var>
Definition: Open a PDOS file for shared access

The SOPEN statement opens a file for shared random access and returns the FILE ID in <var>. All subsequent access to the file is with the FILE ID. A file opened for shared access can be opened by another task. This does not make a new entry in the file slot table and hence, concurrent accesses need to use the LOCK and UNLOCK statements to ensure data integrity. In addition, the same pointer is used by all tasks accessing the file. Hence, a LOCK and POSITION should be used to access data.

The END-OF-FILE marker on a shared file is changed only when the file has been extended. All data transfers are buffered through a channel buffer; data movement to and from the disk is by full sectors.

(FDATA has BASIC numbers 0 to 100 with
 squares and cubes.)

```
LIST
 10  SOPEN "FDATA",FILID
 20  I=INT[100*RND]
 30  BINARY 0,FILID;4,I,18,0;3,J,K,L  !LOCK
 40  PRINT I,J;K;L
 50  GOTO 20
RUN
 62            62 3844 238328
 1             1 1 1
 15            15 225 3375
 14            14 196 2744
 48            48 2304 110592
 41            41 1681 68921
 35            35 1225 42875
 19            19 361 6859
 74            74 5476 405224
 8             8 64 512
 90            90 8100 729000
 93            93 8649 804357
```

10.92 Statement: SPOOL

```
    Format:     SPOOL <exp1> TO <exp2>
                SPOOL <exp1> TO <string>
                SPOOL <exp1>
Definition:     Send console outputs to PDOS file
```

The SPOOL statement specifies a UNIT number with <exp1> and a FILE ID with <exp2> or file name with <string>. When that particular UNIT is selected, all console outputs are written to the selected file. The SPOOL statement is useful in saving output when other peripherals are busy.

If only one expression follows the SPOOL statement, then only the SPOOL UNIT is changed. Thus, a SPOOL 0 temporarily disables spooling.

```
LIST
 10  SPOOL 3 TO "TEMP"
 20  UNIT 3
 30  FOR I=1 TO 5
 40   PRINT I;" SQUARED =";I*I
 50  NEXT I
 60  UNIT 1
RUN
 1 SQUARED = 1
 2 SQUARED = 4
 3 SQUARED = 9
 4 SQUARED = 16
 5 SQUARED = 25

STOP AT 60
DISPLAY "TEMP"
 1 SQUARED = 1



 2 SQUARED = 4
 3 SQUARED = 9
 4 SQUARED = 16
 5 SQUARED = 25

*READY
```

10.93 Function: SQR

```
    Format:     SQR <exp>
Definition:     Returns square root of <exp>
```

The SQR function returns the square root value of a non-negative <exp>. Given a good first approximation to the square root, the following Newton formula requires only four iterations to achieve eleven digits of accuracy:

```
X[i+1] = ( X[i] + N / X[i] ) / 2
```

where N is the <exp> and X[i] is the approximate square root.

```
LIST
 10  FOR I=1 TO 10
 20   PRINT "THE SQUARE ROOT OF";I;" IS";SQR[I]
 30  NEXT I
RUN
THE SQUARE ROOT OF 1 IS 1
THE SQUARE ROOT OF 2 IS 1.4142136
THE SQUARE ROOT OF 3 IS 1.7320508
THE SQUARE ROOT OF 4 IS 2
THE SQUARE ROOT OF 5 IS 2.236068
THE SQUARE ROOT OF 6 IS 2.4494897
THE SQUARE ROOT OF 7 IS 2.6457513
THE SQUARE ROOT OF 8 IS 2.8284271
THE SQUARE ROOT OF 9 IS 3
THE SQUARE ROOT OF 10 IS 3.1622777

STOP AT 30
```

10.94 Function: SRH

Format: SRH[<string1>,<string2>]
Definition: Returns position of <string1> in <string2>

The SRH function searches for <string1> in <string2> and returns the number of the start character of the first occurrence. If the string was not found, a zero is returned.

```
LIST
 10  DIM A[10],B[10]
 20  $B[0]="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
 30  INPUT $A[0];
 40  I=SRH[$A[0],$B[0]]
 50  IF I: PRINT " WAS FOUND AT POSITION";I
 60   ELSE PRINT " WAS NOT FOUND"
 70  GOTO 30
RUN
:ABC WAS FOUND AT POSITION 1
:PQR WAS FOUND AT POSITION 16
:STV WAS NOT FOUND
:
```

10.95 Command: STACK

Format: STACK
Definition: List user stack entries

The STACK command lists to the user console all entries in the GOSUB stack. The entries are listed in order from the first call to last call.

```
LIST
 10  GOSUB 20
 20  GOSUB 30
 30  GOSUB 40
 40  GOSUB 50
 50  GOSUB 60
 60  STOP
RUN

STOP AT 60
STACK
#10
#20
#30
#40
#50
*READY
```

10.96 Statement: STOP

Format: STOP
Definition: Stop program execution

The STOP statement halts program execution and saves the next line number for the continue command (^C).

```
LIST
 10  PRINT "LINE 10"
 20  STOP
 30  PRINT "LINE 30"
RUN
LINE 10

STOP AT 20
CONT
LINE 30

STOP AT 30
```

10.97 Statement: SWAP

Format: SWAP
Definition: Swap to next task

The SWAP command immediately swaps to the next task. This is useful when executing in a tight loop waiting for some event to occur. Wasted CPU cycles can then be used by other tasks.

```
 100  REM WAIT FOR EVENT 30
 110  IF EVF[30]: SWAP : GOTO 100
```

10.98 Function: SYS

Format: SYS <exp>
Definition: Returns value of <exp> system variable

The SYS function returns system parameters as selected by <exp>. SYS[0] through SYS[36] are predefined parameters, while SYS[16] returns elements of the task control block.

Predefined variables are as follows:

SYS[0] = HELP FLAG	
SYS[1] = LAST ERROR #	
SYS[2] = LAST ERROR LINE #	
SYS[3] = DIGITS	
SYS[4] = AUTO INCREMENT	
SYS[5] = OUTPUT UNIT #	
SYS[6] = OUTPUT COLUMN COUNTER	
SYS[7] = LAST RECORD LENGTH	
SYS[8] = COM[ ] SIZE	
SYS[9] = (R9) TASK CONTROL BLOCK POINTER	Read Only
SYS[10] = INPUT PORT #	
SYS[11] = ASSIGNED INPUT MESSAGE POINTER	
SYS[12] = ASSIGNED INPUT FILE ID	
SYS[13] = RANDOM SEED	
SYS[14] = FOR/NEXT STACK SIZE	
SYS[15] = GOSUB STACK SIZE	
SYS[16] = UNIT 1 CRU BASE	
SYS[17] = UNIT 2 CRU BASE	
SYS[18] = CURRENT 'BASE' CRU BASE	
SYS[19] = SYSTEM DISK/DIRECTORY LEVEL	
SYS[20] = BEGINNING OF USER PROGRAM	Read Only
SYS[21] = STATEMENT DEFINITION TABLE	Read Only
SYS[22] = VARIABLE NAME TABLE	Read Only
SYS[23] = VARIABLE DEFINITION TABLE	Read Only
SYS[24] = NEXT VARIABLE DEFINITION	Read Only
SYS[25] = NEXT VARIABLE STORAGE	Read Only
SYS[26] = GOSUB STACK	Read Only
SYS[27] = FOR/NEXT STACK	Read Only
SYS[28] = END USER STORAGE	
SYS[29] = END USER MEMORY	Read Only
SYS[30] = BASIC VARIABLE LENGTH	Read Only
SYS[31] = GOSUB STACK POINTER	Read Only
SYS[32] = USER FUNCTION LINK	Read Only
SYS[33] = REMARK FLAG	
SYS[34] = EXTERNAL TABLE SIZE	
SYS[35] = EXTERNAL TABLE ADDRESS	Read Only
SYS[36] = CURRENT TASK NUMBER	Read Only

```
LIST
 10  ERROR 100
 20  I=10/0
 100  PRINT "ERROR";SYS[1];" AT LINE";SYS[2]
RUN

ERROR 28 AT LINE 20

LIST
 10  FOR I=0 TO 36
 20   PRINT "SYS[";I;" ]=";#SYS[I]
 30  NEXT I
RUN
SYS[ 0 ]=0000
SYS[ 1 ]=0000
SYS[ 2 ]=0000
SYS[ 3 ]=0008
SYS[ 4 ]=000A
SYS[ 5 ]=0003
SYS[ 6 ]=0009
SYS[ 7 ]=0000
SYS[ 8 ]=000A
SYS[ 9 ]=6020
SYS[ 10 ]=0001
SYS[ 11 ]=0000
SYS[ 12 ]=0000
SYS[ 13 ]=0000
SYS[ 14 ]=0008
SYS[ 15 ]=0014
SYS[ 16 ]=0080
SYS[ 17 ]=0000
SYS[ 18 ]=0000
SYS[ 19 ]=0401
SYS[ 20 ]=62F6
SYS[ 21 ]=631A
SYS[ 22 ]=6328
SYS[ 23 ]=6334
SYS[ 24 ]=633A
SYS[ 25 ]=DEB0
SYS[ 26 ]=DEF6
SYS[ 27 ]=DF46
SYS[ 28 ]=E000
SYS[ 29 ]=E000
SYS[ 30 ]=0006
SYS[ 31 ]=DEF6
SYS[ 32 ]=0000
SYS[ 33 ]=0000
SYS[ 34 ]=0014
SYS[ 35 ]=DFD8
SYS[ 36 ]=0000

STOP AT 30
```

10.99 Function: TAN

Format: TAN <exp>
Definition: Returns tangent of radian <exp>

The TAN function returns the tangent of <exp>. The argument is in radians. To convert degrees to radians, multiply the number of degrees by 0.0174533.

```
LIST
 10  FOR I=0 TO 4*ATN 1 STEP 0.2
 20   PRINT TAN I;: X=TAN[I]*4+40
 30   IF X>10: PRINT  TAB X;"*";
 40   PRINT
 50  NEXT I
RUN
 0                                      *
 0.20271004                             *
 0.42279322                              *
 0.68413681                               *
 1.0296386                                  *
 1.5574077                                    *
 2.5721516                                        *
 5.7978837
 -34.232533
 -4.2862617           *
 -2.1850399                    *
 -1.3738231                       *
 -0.91601429                        *
 -0.60159661                         *
 -0.35552983                          *
 -0.14254654                           *

STOP AT 50
```

10.100 Statement: THEN

Format: THEN <statement>
Definition: A TRUE precondition to a line execution

The THEN statement precedes any BASIC statement and continues execution of the program line only if the ELSE FLAG is TRUE. The ELSE FLAG is set FALSE whenever an IF statement is executed. If the IF statement evaluates TRUE, the ELSE FLAG is set TRUE. The flag remains set or reset until another IF statement is executed. Hence, multiple line blocks can be executed or ignored, depending upon how the ELSE FLAG is set.

Program lines beginning with THEN are indented two spaces, when listed.

```
LIST
 10  IF 1<2
 20    THEN PRINT "YES"
 30    ELSE PRINT "NO"
RUN
YES

STOP AT 30
```

10.101 Function: TIC

Format: TIC <exp>
Definition: Returns current tic value less <exp>

The TIC function returns the value of the two word timer less the value of <exp>. The timer is incremented 125 times a second. Hence, one tic equals 1/125 of a second. To mark elapsed time, a variable is assigned TIC[0]. At any time thereafter, TIC of the variable gives the elapsed time in 1/125 second intervals.

```
LIST
 10  T=TIC 0
 20  FOR I=1 TO 1000
 30  NEXT I
 40  T=TIC T
 50  PRINT "FOR LOOP TIME =";T/125;" SECONDS"
RUN
FOR LOOP TIME = 1.12 SECONDS

STOP AT 50
```

10.102 Statement: TIME

Format: TIME
TIME <exp1>{,<exp2>{,<exp3>}}
TIME <string>
Definition: Set or read system time

The TIME statement reads, sets, or displays the system time.

TIME without any parameters displays to the user console an eight character string in the format "HR:MN:SC".

If the parameter of TIME is a string variable, then the same eight character string is stored in the variable.

If an expression <exp1> follows the TIME statement, it is evaluated and used to set the hours of the system clock. A subsequent expression <exp2>, sets the minutes, while expression <exp3> sets the seconds.

```
TIME
19:05:50
TIME 20,30,0
TIME $A[0]
;$A[0];20:30:05
```

10.103 Statement: TRACE

Format: TRACE <exp> {,<var>}
Definition: Set trace options

The TRACE statement is used to monitor program assignments and transfers for debugging purposes.

TRACE 1 is the variable trace mode. All program numeric assignments are output to the user console after the assignment has been made. If an optional variable follows the trace type, then only a single variable is traced. Otherwise, all numeric assignments are shown in the trace. The line number of the assignment is first listed followed by the variable name, an equal sign, and finally the new value. If the variable is a dimensioned variable, then a left bracket follows the variable name.

TRACE 2 is the transfer trace mode. It outputs to the user console any program transfer due to the execution of a GOSUB, ERROR, INPUT help, POP, GOTO, or RETURN statement. The only transfer not listed is the SKIP statement. The first number indicates the point of origin, while the number following the '=>' is the destination of the transfer.

TRACE 4 is the line trace mode. Every line is displayed to the console before it is executed. All other trace outputs follow the listed line.

Any combination of the above trace modes are put in effect at the same time by adding the trace values. For example, transfer and variable trace would be active with TRACE 3. The variable option is only for TRACE 1 and resets every time a TRACE command is executed.

```
LIST
 10  DIM TEMP[10]
 20  I=INT[10*RND]: GOSUB 100
 30  IF FLAG<2: GOTO 20
 40  STOP
 100  COUNT=COUNT+1: IF TEMP[I]<>0: RETURN -1
 110  TEMP[I]=COUNT: COUNT=0: FLAG=FLAG+1
 120  RETURN
TRACE 1
RUN                   TRACE 3
20 I=7                RUN
100 COUNT=1           20 I=3
110 TEMP[=1           20 => 100
110 COUNT=0           100 COUNT=1
110 FLAG=1            110 TEMP[=1
20 I=7                110 COUNT=0
100 COUNT=1           110 FLAG=1
20 I=4                120 => 20
100 COUNT=2           30 => 20
110 TEMP[=2           20 I=6
110 COUNT=0           20 => 100
110 FLAG=2            100 COUNT=1
STOP AT 40            110 TEMP[=1
                      110 COUNT=0
TRACE 2               110 FLAG=2
RUN                   120 => 20
20 => 100             STOP AT 40
120 => 20
30 => 20              TRACE 1,FLAG
20 => 100             RUN
120 => 20             110 FLAG=1
STOP AT 40            110 FLAG=2
                      STOP AT 40
TRACE 4
RUN
 10  DIM TEMP[10]
 20  I=INT[10*RND]: GOSUB 100
 100  COUNT=COUNT+1: IF TEMP[I]<>0: RETURN -1
 110  TEMP[I]=COUNT: COUNT=0: FLAG=FLAG+1
 120  RETURN
 30  IF FLAG<2: GOTO 20
 20  I=INT[10*RND]: GOSUB 100
 100  COUNT=COUNT+1: IF TEMP[I]<>0: RETURN -1
 110  TEMP[I]=COUNT: COUNT=0: FLAG=FLAG+1
 120  RETURN
 30  IF FLAG<2: GOTO 20
 40  STOP
STOP AT 40
```

10.104 Function: TSK

Format: TSK <exp>
Definition: Return the task <exp> status

The TSK function returns the status of task <exp>. If TSK[<exp>] is zero, then task <exp> is not in the task list. If TSK[<exp>] is positive, then task <exp> is executing and TSK[<exp>] is its task time. If TSK[<exp>] is negative, then task <exp> is suspended pending event -TSK[<exp>].

```
.LT
TASK  PAGE  TIME    TB    WS    PC    SR ...

*0/0   0     3    >6020 >619A >04C2 >1005...
 1/0   0    -97   >DC20 >DD9A >03F0 >C605...
.EX
*READY
LIST
 10  FOR I=0 TO 5
 20   PRINT "TASK";I;" STATUS =";TSK[I]
 30  NEXT I
RUN
TASK 0 STATUS = 3
TASK 1 STATUS = -97
TASK 2 STATUS = 0
TASK 3 STATUS = 0
TASK 4 STATUS = 0
TASK 5 STATUS = 0
```

10.105 Statement: UNIT

Format: UNIT <exp>
Definition: Direct console outputs

The UNIT statement assigns ASCII output to the device indicated by <exp>. UNIT 1 is the system console CRT. UNIT 2 is the auxiliary output number.

Each bit of the UNIT variable selects a different output device. Various bits are assigned to different devices or files with the SPOOL command.

```
STOP AT 30

BAUD -2,1200      Set UNIT 2 to >0180 at 1200 baud
UNIT 3            Send output to CRT and AUX port
```

10.106 Statement: WAIT

Format: WAIT <exp>
Definition: Suspend task pending event <exp>

The WAIT command suspends the user task until the event specified by <exp> occurs. There are 127 events defined in PDOS. The first 15 (1-15) are hardware events, while events 16 through 127 are software events. (Event 0 is ignored.)

```
100. BASE 00180H  !POINT TO AUX 9902
110  WAIT 5  !SUSPEND TASK
120  CR=CRF[8]  !READ CHARACTER
130  CRB[18]=1  !ACKNOWLEDGE INTERRUPT
....
```

```
1-15    Hardware events
16-127 Software events
```

A task that has been suspended does not receive any CPU cycles until the event occurs. When the event occurs, the task begins executing the next statement after the WAIT statement. This is immediate, if it was a hardware event. Otherwise, the task continues execution during the normal swapping functions of PDOS.

A suspended task is indicated in the LIST TASK (LT) command by a minus event number being listed for the task time parameter. When the event does occur, the time parameter is restored.

```
.LT
 TASK  PAGE  TIME    TB    WS    PC    SR ...

 *0/0   0     3    >42A2 >441C >0654 >D40F ...
  1/0   0    -30   >4AA2 >4A82 >1040 >D00F ...
  2/0   0    -5    >52A2 >5282 >292E >C40F ...
```

Hardware events are enabled by overwriting the appropriate interrupt vector with the workspace and address of the event processor. Also, the interrupt mask bit on the 9901 is set to one, enabling the interrupt. Software events are indicated by a single bit being set or reset in an event list.

If more than one task is suspended on the same event, only the lowest numbered task is awakened for all hardware events. For software events, however, all tasks suspended on the event begin executing. For a hardware event, further interrupts on the event level are disabled at the system TMS9901 by setting the interrupt mask bit to zero. The system interrupt mask is not affected. Software event flags are not reset and must be processed by the event routine.

```
200  WAIT 30  !SUSPEND UPON EVENT 30
210  EVENT -30  !ACKNOWLEDGE EVENT
....
```

See 5.2.16 XSUI - SUSPEND UNTIL INTERRUPT