CHAPTER 9

PDOS BASIC

Chapter 9 introduces you to PDOS BASIC. Although most standard Dartmouth BASIC verbs are the same, many extensions have been added to support industrial, scientific, and business applications. File management and context strings are easy to understand and very versatile.

(CHAPTER 9 PDOS BASIC continued)

9.1 BASIC PRIMER

Interactive interpreters

Microcomputer interpreters are generally slow and not competitive in performance with comparable compilers. Despite this disadvantage, BASIC interpreters have been implemented on almost every microcomputer available today and are widely used for both business and scientific applications. This wide acceptance is due mainly to the interactive nature of interpreters.

Reverse Polish pseudo source tokens

The PDOS BASIC interpreter combines performance and interaction with a unique approach. PDOS BASIC pseudo source tokens are parsed during program entry and not at execution time. In other words, the evaluator executes serially through the tokens, since they are stored in correct Reverse Polish order. This is immediately evident when a program is listed using the LISTRP command.

PDOS BASIC features:

- -Meaningful variable names
- -Multi-statement recursive functions
- -Function local variables
- -Extensive line editing commands
- -Fast 48-bit floating point arithmetic
- -11 digit accuracy
- -Context oriented string primitives
- -Full disk file interface primitives
- -Standalone run module support
- -CRU instruction primitives
- -Assembly language linkage
- -Color graphic primitives
- -Speech primitives
- -Variable, transfer, and execution trace
- -Program chaining
- -Formatted print commands
- -Inter-task communication arrays
- -Memory functions
- -Time and date commands
- -Logical operators
- -Suspend task command
- -Set and test event commands

9.1.1 AN EXAMPLE

The system of two simultaneous linear equations in two variables, ax + by = c, and dx + ey = f, can be solved if (ae - bd) is not equal to zero. The solution is given by:

```
ax + by = c
dx + ey = f
```

```
     ce - bf         af - cd
x = -------     y = -------
     ae - bd         ae - bd
```

If (ae - bd) = 0, there is either no solution or there are infinitely many, but there is no unique solution. Study the program example to the right carefully. In most cases, the purpose of each line in the program is self-evident.

```
LIST
 10  READ A,B,D,E
 20  G=A*E-B*D
 30  IF G=0: GOTO 90
 40  READ C,F
 50  X=(C*E-B*F)/G
 60  Y=(A*F-C*D)/G
 70  PRINT X,Y
 80  GOTO 40
 90  PRINT "NO UNIQUE SOLUTION"
 100  DATA 1,2,4
 110  DATA 2,-7,5
 120  DATA 1,3,4,-7
 130  STOP
RUN
 4              -5.5
 0.66666667     0.16666667
 -3.6666667     3.83333333

*ERROR 21 AT 40
```

Several things are immediately apparent from this sample program. First, the program uses only capital letters. Second, each line of the program begins with a number. These numbers are called line numbers and serve to identify the lines, each of which is called a statement.

A program is made up of statements that are executed by the computer. A program can be entered in any order and the computer sorts out and edits the statements into the order specified by their line numbers.

Third, each statement starts, after its line number, with an English word unless it is an assignment statement. The English word denotes the type of statement. Spaces are used to delimit variables and expressions but are not stored with the program.

With this preface, let us examine the program step by step. The first statement, 10, is a READ statement. It must be accompanied by one or more DATA statements. When the computer encounters a READ statement while executing your program, it assigns values to the variables listed after the READ statement according to the next available values in the DATA statement.

In the example, variable A of line 10 is assigned the value of 1 from the DATA statement of line 100. Similarly, B is assigned a 2 and D a 4. At this point, the available data in statement 100 is exhausted. The computer moves on to line 110 and assigns variable E the value of 2.

Statement 20 is an assignment statement. The variable G is assigned by the computer the results of the expression (ae - bd). (The '*' is used for multiplication.) In general, an assignment statement gives the variable on the left side of the equal sign the value of the expression on the right side.

(9.1.1 AN EXAMPLE continued)

If G is equal to zero, the system has no unique solution. Therefore, line 30 asks the question, "is G equal to zero?" The statement is an 'IF' statement and sets an internal flag TRUE for 'yes' and FALSE for 'no'. If the flag is TRUE, then the computer continues executing the statement. The computer would 'GOTO' line 90 and print on your console 'NO UNIQUE SOLUTION'. Otherwise, the computer moves immediately to the next statement.

The computer now reads two more values from the DATA statements, namely -7 and 5, and assigns them to variables C and F respectively. The computer can now solve the system of equations. Note that parentheses must be used to indicate that (C*E-B*F) is divided by G. Without parentheses, only (B*F) would be divided by G, which results in a wrong answer.

The computer prints the two results in line 70. Line 80 directs execution back to line 40. If there are additional numbers in the DATA statements, then another system of equations is solved. This continues until there are no more numbers in DATA statements, at which time an error is reported.

Why is the program numbered by tens? The answer is that the particular choice of line numbers is arbitrary, as long as the statements are numbered in the order that the computer is to follow in executing the program. The statements could have been numbered 1, 2, 3, ... 13. This is not recommended since later insertions would be impossible if you forget a line when entering the program.

9.1.2 EXPRESSIONS

The computer can perform many arithmetic operations: it can add, subtract, multiply, divide, extract square roots, raise a number to a power, find the sine of an angle, etc. This is the primary function of a computer.

Expressions are similar to those used in standard mathematical calculations, with the exception that all BASIC expressions must be written on a single line. Operator precedence is observed in formulating expressions. If you enter 'A + B * C ^D', the computer raises C to the power D, multiplies the result by B, and then adds A to the product. If this is not the intended order, then parentheses must be used to group the operations.

(9.1.2 EXPRESSIONS continued)

The order of priorities is summarized as follows:

1. The expression inside the parentheses is evaluated before the parenthesized quantity is used in further computations.

   (10+2)*5 = 60

2. In the absence of parentheses, the computer first evaluates the unary minus operator followed by powers, division, multiplication, subtraction, and finally addition.

   -10+2*2^3 = 6

3. In the absence of parentheses in an expression involving operations of the same priority, (multiplication - division, addition - subtraction), the operations are performed from left to right.

   1+2-3+4*6/3 = 8

In addition to the six arithmetic operators, many intrinsic mathematical and system functions are available. These functions may be used in the place of any operand of an expression. Some have two arguments. All return a numeric value.

Mathematical functions:

ABS(X),ATN(X),COS(X),EXP(X)
FRA(X),INP(X),INT(X),LOG(X)
SGN(X),SIN(X),SQR(X),TAN(X)

System functions:

ADR(X),BIT(I,X),CRB(X),CRF(X)
EVF(X),KEY(X),MEM(X),MEMP(X)
MEMW(X),SYS(X),TIC(X),TSK(X)

String functions:

LEN(X),MCH(X,Y),NCH(X),SRH(X,Y)

A number may be positive or negative with up to 11 digits of precision. All of the following are valid numbers:

2	-3.675	3.1415926
1234567	-76.5432343	0.00123.

Further flexibility is gained by using the letter 'E', which stands for 'times ten to the power'. The following are equivalent:

.00123456789	1234567.89E-9
.123456789E-2	1234.56789E-6.

Ten million can be written as 1E7 or 1E+7, not as E7. Numbers are stored in either integer or floating point format. Numbers range from approximately 1E-78 to 1E76.

A variable is a quantity whose value can be changed by BASIC instructions. There are basically two types of variables; simple and dimensioned. A simple variable begins with an alpha character followed by any number of alpha-numeric characters or underlines (_). A dimensioned variable is a simple variable followed by parenthesized subscripts. The RUN and CLEAR statements initialize all variables to zero.

APPLE=PAY_DAY
DOG[TYPE,AGE]

9.1.3 LOOPS

Frequently it is necessary to write a program in which one or more portions of the program are performed not just once, but a number of times, perhaps with slight changes each time. In order to write the simplest program in which the portion to be repeated is written just once, you use the programming device known as a loop.

The loop structure is illustrated by a program which prints the first 100 positive integers and their square roots. Without a loop, the first program has 101 statements and looks like:

```
10  PRINT 1,SQR(1)
20  PRINT 2,SQR(2)
30  PRINT 3,SQR(3)
....
990  PRINT 99,SQR(99)
1000  PRINT 100,SQR(100)
1010  STOP
```

Using the 'IF - GOTO' type of loop, the same program can be shortened considerably to:

```
10  X=1
20  PRINT X,SQR(X)
30  X=X+1
40  IF X<=100: GOTO 20
50  STOP
```

Statement 10 initializes the variable X to 1. Line 20 prints both X and its square root. Line 30 increments X and line 40 checks to see if X is less than or equal to 100. The program loops back to line 20 each time through, until X is greater than 100.

All loops have these same characteristics: 1) initialization (line 10, 2) body (line 20), 3) modification (line 30), and 4) an exit test (line 40). Because loops are so important and arise so often, BASIC has included in it the FOR and NEXT statements. These statements simplify the program to:

```
10 FOR X=1 TO 100
20 PRINT X,SQR(X)
30 NEXT X
40 STOP
```

(9.1.3 LOOPS continued)

Line 10 initializes X to 1 and sets the limit to 100. Line 30 increments X by 1 and checks against the limit. If X is less than or equal to the limit, execution returns immediately to the next statement after the FOR statement. (This may be on the same line.) When X exceeds the limit, execution drops through to the statement following the NEXT.

The step value defaults to one, but may be changed to any value with the STEP parameter. The table could be printed in reverse order by rewriting line 10 as:

```
10 FOR X=100 TO 1 STEP -1
```

For a positive step size, the loop continues as long as the control variable is algebraically less than or equal to the final value. For a negative step size, the loop continues as long as the control variable is greater than or equal to the final value.

If the initial value is greater than the final value (less than for negative step size), then the body of the loop is not executed at all. Execution continues with the statement immediately following the corresponding NEXT statement. This is call a pretest.

It is useful to have loops within loops. These are called nested loops. They must actually be nested and cross outside the scope of each loop.

```
ALLOWED                 ALLOWED

FOR X                   FOR X
  FOR Y                  FOR Y
  NEXT Y                  FOR Z
NEXT X                    NEXT Z
                          FOR W
                          NEXT W
NOT ALLOWED              NEXT Y
                         FOR Z
FOR X                    NEXT Z
FOR Y                   NEXT X
NEXT X
NEXT Y
```

9.1.4 ARRAYS

In addition to the ordinary or simple variables, there are dimensioned variables which allow you to reference many variables with the same variable name. These variables use subscripts to reference sub-elements such as the coefficients of a polynomial [a0, a1, a2,...] or the elements of a matrix [i,j].

A dimensioned variable consists of a simple variable followed by the subscripts in brackets or parentheses. You might use A(0), A(1), A(2), etc. for the coefficients of a polynomial and B(1,1), B(1,2), etc. for the elements of a matrix.

An array must be dimensioned in a program before it is used. The DIM statement reserves memory for the elements of an array. An array can have up to 7 dimensions. A simple and dimensioned variable of the same name do not reference the same element.

Array subscripts begin with zero. An array dimensioned as DIM A[10,10] has 11 x 11 or 121 elements. (A[0,0], A[0,1], ... , A[10,10].)

The program to the right uses both a single and a double subscripted array. The program computes the total sales for each of five salesman, all of whom sell the same three products.

The array P gives the price per item of the three products and the array S tells how many items of each product each man sold. You can see from the program that product #1 sells for $1.25 per item, #2 for $4.30, and #3 for $2.50. Salesman #1 sold 40 items of the first product, 10 of the second, 35 of the third, and so on.

The program reads in the price array with lines 20 through 40. Lines 50 through 90 read each man's sales. Lines 100 through 160 process and print total sales.

Array elements are stored in memory in consecutive memory locations. The rightmost subscript changes the fastest. Hence, the two dimensional array, A[1,2] is stored as:

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

```
LIST
 10  DIM P[3],S[3,5]
 20  FOR I=1 TO 3
 30   READ P[I]
 40  NEXT I
 50  FOR I=1 TO 3
 60   FOR J=1 TO 5
 70    READ S[I,J]
 80   NEXT J
 90  NEXT I
 100  FOR J=1 TO 5
 110   S=0
 120   FOR I=1 TO 3
 130    S=S+P[I]*S[I,J]
 140   NEXT I
 150   PRINT "TOTAL SALES FOR SALESMAN";J;" = $";S
 160  NEXT J
 170  STOP
 500  DATA 1.25,4.3.2.5
 510  DATA 40,20,37
 520  DATA 29,42,10
 530  DATA 16,3,21
 540  DATA 8,35,47
 550  DATA 29,16,33
RUN
TOTAL SALES FOR SALESMAN 1 = $ 180.5
TOTAL SALES FOR SALESMAN 2 = $ 211.3
TOTAL SALES FOR SALESMAN 3 = $ 131.65
TOTAL SALES FOR SALESMAN 4 = $ 166.55
TOTAL SALES FOR SALESMAN 5 = $ 169.4

STOP AT 170
```

9.2 BASIC DEFINITIONS

9.2.1 BASIC COMMANDS

Single non-program instructions

A BASIC command is a single instruction to the interpreter, such as NEW. Such items cannot be entered into a program and generally refer to the BASIC system as a whole. No more than one command can appear on a line.

The commands LIST and LISTRP display on the console the current program, with statement numbers in ascending order. The NEW command clears the user work area, destroys the current program, and initializes all pointers and buffers. The RUN command begins program execution at the lowest line number.

PDOS BASIC commands include:

FILES	Print disk directory
LIST	List user program
LISTRP	List user program in Reverse Polish
NEW	Clear user work area
SIZE	Print memory usage
STACK	Print user GOSUB stack contents

9.2.2 STATEMENTS

Program instructions

A BASIC statement is also a single instruction to the interpreter. Statements can begin with an signed line number ranging from -32767 to 32768 (excluding 0), followed by an instruction word, followed by any expression(s) needed by the instruction, followed by perhaps comments, and finally, the statement terminator (<carriage return> or :).

Multiple statements can appear on one line by separating them with a single colon (:). If no line number is given, the statement is immediately executed. This is referred to as keyboard mode.

```
10  A=1: B=4: GOSUB 100
```

BASIC statements may be entered by the programmer in any order. They are sorted into ascending order according to statement number. To insert a line, for example, between the statements numbered 20 and 30, give the new statement a line number greater than 20 and less than 30. To replace a line, enter the new statement with the same line number. To delete a line, enter the statement number only.

```
50 NEXT I
20 FOR I=1 TO 10
30 PRINT I
LIST
 20  FOR I=1 TO 10
 30    PRINT I
 50  NEXT I
```

9.2.3 PDOS BASIC STATEMENT SUMMARY

Control

ELSE	Execute on FALSE condition flag
ERROR	Error trapping
ESCAPE	Allow break character
FNPOP	Pop function stack
FOR	Loop header
GOTO	Unconditional program transfer
GOSUB	Subroutine call
IF	Set condition flag
NEXT	Loop foot
NOESC	Disable break character
ON	Computed GOTO, GOSUB
POP	Decrement GOSUB stack
RETURN	Subroutine exit
RUN	Begin execution or chain
THEN	Execute on TRUE condition flag
SKIP	Conditional jump
STOP	Stop
SWAP	Swap to next task

Interrupts and Task communication

COM	Common array
EVENT	Set software event
EVF	Test event flag
MAIL	Global array
WAIT	Suspend task pending event

Evaluation

BIT	Variable bit assignment
CALL	Function or assembly subroutine call
CRB	CRU bit assignment
CRF	CRU multiple bit assignment
LET	Variable assignment
MEM	Memory byte assignment
MEMW	Memory word assignment
MEMP	Memory page assignment
READ	Variable assignment

Interpreter

BYE	Exit to PDOS
CLEAR	Clear variable space
PURGE	Delete program lines
TRACE	Monitor program execution

(9.2.3 PDOS BASIC STATEMENT SUMMARY continued)

Definitions

DATA	Program data storage
DEFN	Function header
DIM	Array declaration
EQUATE	Variable associations
EXTERNAL	External command table
FNEND	Function foot
GLOBAL	Declare variable address
LABEL	Define line variable
LOCAL	Function local variable
REM	Remark
RESTORE	Move DATA pointer

INPUT/OUTPUT

BASE	CRU base
BAUD	Port initialization
DATE	Read/set system date
INPUT	Read keyboard input
PRINT	Data output
TIME	Read/set system time
UNIT	Output selection

DISK I/O

CLOSE	Close file
DEFINE	Define file
DELETE	Delete file
DISPLAY	Display file to console
FILE	Select, read, write, position
GOPEN	Open file for read only
LOAD	Load program
OPEN	Open file for sequential access
PDOS	Read PDOS parameters
RENAME	Rename file
RESET	Reset files by task/disk
ROPEN	Open file for random access
SOPEN	Open file for shared access
SAVEB	Save program in pseudo source tokens
SAVE	Save program in ASCII format
SPOOL	Direct output to file

String

Assignment	Convert binary to decimal
Pick	Convert decimal to binary
Replace	Convert decimal to hex
Concatenate	Convert hex to decimal
Search	Convert hex string
Match	Insert
Length	Delete

9.2.4 CONSTANTS

An arithmetic constant in BASIC represents a numeric value. All BASIC numbers are stored in 48 bits (3 words) of memory. This gives 11 digits of precision with a range of approximately 10 raised to the plus or minus 74th power.

```
CONSTANT RANGE = 1E-74 TO 1E74
```

The internal storage format varies and is transparent to your program. Floating point numbers consist of a sign bit, 7 bits of exponent (biased by 64), and 40 bits of fraction. The implied binary point is immediately to the left of the MSB of the 40-bit fraction. Integers are stored with the first word equal to zero and the second word set to the 16 bit 2's complement integer. The third word is undefined. When possible, BASIC stores numbers in the integer format to improve execution speeds.

```
1.0 = >4110 >0000 >0000
  1 = >0000 >0001 >0000
```

HEX constants have a decimal leading digit (0-9) followed by the hexadecimal constant and the letter "H". Blanks are not allowed in a hex constant.

```
II=0FFEH
$HEX=#-2
PRINT II;#-3$HEX; -2 FFFE FFFD
```

String constants are represented by a string of characters enclosed in double or single quotes. This also applies to string constants in DATA statements.

```
A=10
$A="10"
DATA 1, "ONE",'"quote"'
```

9.2.5 VARIABLES

A variable is a quantity whose value is changed by BASIC instructions. There are basically two types of variables: simple and dimensioned. There are two modes in which these variables can be used, namely numeric and string.

A simple variable begins with an alpha character (A-Z) followed by any number of alpha-numeric characters or underlines (_). Dimensioned variables are simple variables followed by up to 7 dimensions enclosed in parentheses. These variables are arrays which group numbers together in the form of a matrix or list (a vector). Subscripts are used to reference individual elements within an array. Dimensioned variables are not the same as simple variables with the same name. (A[0] is not that same as the simple variables A or A0.)

```
A
APPLE
PAY_DAY
A[10]
MULTI[1,2,3,4]
```

String variables require no formal declaration but are merely simple or subscripted variables preceded by a dollar sign ($). String variables are context defined, which simply means that variables can hold any kind of data and are typed only by the way they are used. Hence, an array can hold character as well as numeric data. (A[0] is identical to $A[0].)

```
$NAME[10]
```

9.2.6 OPERATORS

There are four primary types of operators in PDOS BASIC: logical, relational, algebraic, and string. Relational and logical operators are the following:

Logical operators

LOR	Logical OR
LXOR	Logical exclusive OR
LAND	Logical AND
LNOT	Logical NOT

Relational operators

NOT	Relational NOT
AND	Relational AND
OR	Relational OR
=	Equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
<>	Not equal

Relational operators appear in any arithmetic or string expression and evaluate to zero for FALSE or one for TRUE. Logical operators also appear in arithmetic expressions and return 16-bit signed integers.

Algebraic operators are defined as follows:

Algebraic operators

+	Add
-	Subtract
*	Multiply
/	Divide
^	Raise to the power
-	Unary minus

These operators are used in algebraic expressions and are evaluated in the order of precedence in which they appear in the above table. Operators with the same precedence (e.g., (+,-) or (*,/)) are evaluated from left to right. Parentheses are used to override this order of precedence. The order of precedence with unary operators and exponentiation depends on the form of the expression. If the unary operator is needed to evaluate the exponent, it is used first.

The fourth type of operator is a string operator. These operators include:

String operators

\	Delete or insert
&	Concatenate
#	Convert
%	Byte convert

9.2.7 FUNCTIONS

BASIC functions are of two types, user defined and system defined. User defined functions are added to the program library by the DEFN statement. System functions are predefined and always resident. The system functions include:

ABS	Absolute value
ADR	Memory address
ATN	Arctangent
BIT	Variable bit examine
COS	Cosine
CRB	CRU bit examine
CRF	CRU multiple bit examine
EVF	Test event flag
EXP	Exponential
FRA	Fractional part
INP	Integer part
INT	Greatest integer function
KEY	Input port examine
LEN	String length
LOG	Natural logarithm
MCH	String match
MEM	Memory byte examine
MEMW	Memory word examine
MEMP	Memory page examine
NCH	Numeric value of ASCII character
SGN	Sign function
SIN	Sine
SRH	String search
SQR	Square root
SYS	System parameters
TAN	Tangent
TIC	Clock tics (1/125 second)
TSK	Task status

See 10.20 DEFN and 10.34 FNEND.

9.3 LINE EDITING

Many editing features are included in the PDOS BASIC interpreter. A line buffer is used for program entry. The cursor is moved forward and backward without disturbing the buffer. Facilities are provided for character insertions and deletions as well as rubout and line cancel. A program line is listed to the edit buffer by entering the line number followed by a control E. The line is listed to the screen and the cursor is placed at the end of the line, ready for editing. (A '^' indicates that the control key is held down while the following character is depressed.)

Most of the editing functions are control characters. Some of these include:

^C Continue execution after escape or STOP statement.

^Dn Delete (n) characters beginning at the cursor position.

n^E List into the edit buffer the program line specified by the line number (n). The cursor is positioned at the end of the line ready for editing.

```
100^E
 100 LGC[5]=4*ATN 1-SIN[ERT*CV]_
                               ^
```

^F Forward space 1 character.

^H Backspace 1 character.

^In Insert (n) blanks at the cursor position.

LF Enter current line into program and prompt with next line number.

CR Enter current line into program.

escape Program break character or disregard entered line.

rubout Delete 1 character to the left of the cursor.

When a line is entered into the program, it is immediately parsed for correct Reverse Polish format. If an error is detected, the error number is listed, the line is echoed back to the screen, and the cursor is placed over the offending section.

```
100A(10)=4*10+(AB+DC))
*ERROR 02
100 A(10)=4*10+(AB+DC))
                     ^
```

9.4 BASIC STRINGS

PDOS BASIC strings are context oriented. How data is interpreted depends entirely upon its context within a program. For example, the bit pattern:

```
01000001 01000010 01000011 01000100 01000101 00000000
```

could represent the floating point number 4.1414225, the six 8-bit integers 65, 66, 67, 68, 69, and 0, or the ASCII string 'ABCDE'. Hence, a single variable can be assigned a number and later, a string. An array can contain integers, floating point numbers, and strings all at the same time.

A dollar sign '$' preceding the variable name indicates to the PDOS BASIC interpreter that the content of the variables is to be treated as 8-bit ASCII characters. Strings are stored left justified and delimited by a null character (a zero byte).

A simple variable can hold up to 5 characters plus the null character. Dimensioned variables can hold up to the product of the dimensions times 6 minus 1 (a null character ends the string). Since strings are context oriented, no checking is done by the interpreter for variable overflows.

One additional characteristic of string array variables is that individual bytes within the variable are referenced by following the subscripts with a semicolon and a byte index. The first byte of a string is referenced with index 1.

Strings are stored one ASCII character per byte and are terminated with a null byte. If $A is assigned "HELLO" and A is defined at memory location >D000, then memory would contain the following:

```
ADR[A] >> 4845 4C4C 4F00
```

```
DIM A(2,1)
$A(0,0)="RHINOCEROS"
$A(1,0)="ELEPHANT"
$A(2,0)="GIRAFFE"
;$A(0,0);RHINOCEROS
;$A(1,0);ELEPHANT
;$A(2,0);GIRAFFE
;$A(0,1);EROS
;$A(1,1);NT
;$A(2,1);E
$A(0,1)=$A(2,0)
;$A(0,0);RHINOCGIRAFFE
;$A(0,0;1)RHINOCGIRAFFE
;$A(0,0;2);HINOCGIRAFFE
;$A(0,0;7);GIRAFFE
```

```
LIST
 10  DIM A[2,1]
 20  $A[0,0]="RHINOCEROS"
 30  $A[1,0]="ELEPHANT"
 40  $A[2,0]="GIRAFFE"
 50  $A="HELLO"
 60  B=100
 70  C=3.1415926
RUN

STOP AT 70
;#ADR C;DE78
.MDUMP >DE78,>DEB7
DE78-DE7F  4132 43F6 9A25 0000
DE80-DE87  0064 0000 4845 4C4C A2Cv.%...d..HELL
DE88-DE8F  4F00 0001 0002 0002
DE90-DE97  FFFF 5248 494E 4F43 O.........RHINOC
DE98-DE9F  4552 4F53 0000 454C
DEA0-DEA7  4550 4841 4E54 0000 EROS..ELEPHANT..
DEA8-DEAF  0000 4749 5241 4646
DEB0-DEB7  4500 0000 0000 0000 ..GIRAFFE.......
```

9.4.1 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 the angle brackets are not expanded. The assignment continues byte by byte, until a null character is encountered in the source string. 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 stores 59 characters (10 x 6 - 1).

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

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

9.4.2 STRING EXTRACTION

Characters are extracted from a string by following the string to the right of the equal sign with a comma and an 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, unlike string assignment, ignores all characters, including any nulls, in the source string.

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

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

9.4.3 STRING REPLACEMENT

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 moved to the string variable on the left of the equal sign. A null character is not stored when the transfer is completed.

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

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

9.4.4 STRING CONCATENATION

Strings are concatenated together with 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
```

9.4.5 DELETE CHARACTER

Characters are deleted from a string variable by following the equal sign with a backslash (\) and an expression. The expression specifies how many characters are to be deleted beginning at the string address to the left of the equal sign. 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
```

9.4.6 INSERT CHARACTER

Characters are inserted into a string by following the equal sign with a backslash (\) and a string. Characters are inserted beginning at the string address to the left of the equal sign. If the <string> is null, nothing is inserted.

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

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

9.4.7 CONVERT TO ASCII

A number is converted to a string by assigning it to a string variable. The conversion is format free and uses the current digits size (SYS[3]) in determining the string length and rounding digit. The string is terminated by a null character.

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

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

9.4.8 CONVERT TO ASCII FORMATTED

A number <exp> is converted to a string <string-var> using the format <string>, which follows the equal sign, pound sign. The format string follows the same formatting rules 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
```

9.4.9 CONVERT TO HEX

A number is converted to a four character hex ASCII string by following the equal sign with a pound sign and an 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
```

9.4.10 CONVERT BYTE

Individual bytes may be inserted into a string by following the equal sign with a percent sign and an 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
```

9.4.11 CONVERT STRING

A hexadecimal ASCII string is converted to binary by following the equal sign with a percent sign and a string. Blanks are the only valid non-hex characters allowed and may be used for clarity. Hexadecimal characters are defined as 0-9 and A-F.

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

$A[0;2]=%"2E2E 2A2A"
;$A[0];A..**FGHIJKLMNOPQRSTUVWXYZ
$A[0]=%"41424300"
;$A[0];ABC
```

9.4.12 CONVERT ASCII TO NUMBER

An ASCII string is converted to a binary number by assigning a numeric variable to a string. Since a complete conversion may not be possible, the string can be optionally followed by a comma and a variable to hold the delimiting character. The terminating byte is stored in the first byte of the variable. Hence, if the delimiter variable equals the null string, the conversion was successful.

```
<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 allowed such chaining are %, \, #, and &.

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

9.5 BASIC FILE MANAGEMENT

BASIC supports file read, write, and position. Files are opened in one of four different modes depending upon how they are to be used. Shared files are locked and unlocked for multi-task access. It is your responsibility to block file data into records, although the position statement assists you with fixed record access parameters.

All file access is to the last selected file. With multiple file routines, a select statement must be used to move from one file to the next.

A file must be opened before it can be accessed. The open statement returns the file slot parameter which is used to subsequently select the file. The types of open statements follow:

OPEN Use the OPEN statement for sequential input or output data streams, such as listing to a printer or reading cards from a card reader.

GOPEN Use the GOPEN statement for read only, random access. The file is available for access by other tasks and cannot be written to.

ROPEN Use the ROPEN statement for read/write random access files. The file is not shared and belongs exclusively to your task until it is closed. The CLOSE statement does not write a new end of file unless the file has been extended.

SOPEN Use the SOPEN statement for read/write, shared random access files. Other tasks may also open the file. It is your responsibility to use the lock and unlock statements to resolve task conflicts. There is no automatic record locking in PDOS.

Your program must close all files when done. This allows the operating system to flush all sector buffers and update pointers and dates in the disk directories.

```
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
 ....
```

(9.5 BASIC FILE MANAGEMENT continued)

The FILE statement is the primary file I/O statement and is used to select, read, write, and position within a file. The command expression immediately follows the FILE verb. (Remember, all verbs are delimited by blanks.) Constants may be used but it is recommended that the variables listed to the right be defined and used instead. This makes your program readable. The nine FILE command types are:

```
10 SELECT=1  !FILE SELECT
20 WRITE=2  !FILE WRITE
30 READF=3  !FILE READ
40 POSITION=4  !FILE POSITION
50 LOCK=7  !LOCK FILE
60 UNLOCK=8  !UNLOCK FILE
```

1. FILE 0,<fileid>{,<length>}	Select file and lock task
2. FILE 1,<fileid>{,<length>}	Select file
3. FILE 2,<data>...	Write to file
4. FILE 3,<variable>...	Read from file
5. FILE 4,<length>,<record>,<index>	Position file
6. FILE 5,<string>...	Write line
7. FILE 6,<string variable>...	Read line
8. FILE 7,<fileid>,<code>	File lock
9. FILE 8,<fileid>	File unlock

9.5.1 SELECT AND LOCK TASK

Format: FILE 0,<fileid>,{<length>}

```
SOPEN "DATA:BIN",FID
FILE 0,FID;3,I,J,K
```

The file selected by <fileid> is used for subsequent file access. Variable data length is optionally specified by <length>. The default is 6 bytes. The task is locked while the entire FILE statement is executed. Before another statement is executed, the task lock is cleared. This is used when two users are randomly accessing the same file.

9.5.2 SELECT FILE

Format: FILE1,<fileid>{,<length>}

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

The file selected by <fileid> is used for subsequent file access. Variable data length is optionally changed to <length>. The default is 6 bytes. A new length is in effect until another BASIC verb is executed. (This includes another FILE.)

(9.5 BASIC FILE MANAGEMENT continued)

9.5.3 WRITE TO FILE

Format: FILE 2,<data>...

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

Each expression following the command type is evaluated and written to the last selected file. The data length of each variable is 6 bytes unless changed by a select command within the same FILE statement. The file pointer is updated after each write.

9.5.4 READ FROM FILE

Format: FILE 3,<variable>...

FILE 3,A,B,N[2]

Data is read from the last selected file into each variable following the command type. The data length of each variable is 6 bytes unless changed by a select command within the same FILE statement. The file pointer is updated after each read.

9.5.5 POSITION FILE

Format: FILE 4,<length>,<record>,<index>
FILE 4,<length x record + index>

FILE 4,4*6,I,0

The last selected file's pointer is positioned to a byte index of <length> x <record> + <index>. If the three parameters are used, then no expression can exceed 32767, whereas, a single expression can be any size. <Length> is the record length in bytes. <Record> is the record number, and <index> is a byte displacement into the record.

9.5.6 WRITE LINE

Format: FILE 5,<string>...

FILE 5,"HELLO TURKEY"

The strings following the command type are written to the last selected file. Each string is delimited by a null character. The number of bytes transferred is equal to the length of the string. It is not affected by a FILE select command. The write is independent of the data content. The file pointer is updated after each write operation.

(9.5 BASIC FILE MANAGEMENT continued)

9.5.7 READ LINE

Format: FILE 6,<string variable>...

String data is read from the last selected file into the string variables following the command type. Each read operation is data dependent and terminates upon encountering either a <carriage return> or 132 characters. The <carriage return> is replaced by a null character and all <line feed> characters 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
```

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. Both FILE 5 and FILE 6 are limited to 132 characters.

```
LIST
 100  DIM A[10]
 110  $A[0]="ABCDEFGHIJKLMNOPQRSTUVWXUZ"
 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

ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
STOP AT 220
```

9.5.8 LOCK FILE

Format: FILE 7,<fileid>,<code>

The FILE 7 statement prevents access to a shared file by any other task. The expression <fileid> specifies the file. The variable <code> 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
```

(9.5 BASIC FILE MANAGEMENT continued)

9.5.9 FILE UNLOCK

Format: FILE 8,<fileid>

The FILE 8 statement unlocks a locked shared file so that other tasks can access it.

```
LIST
 10  SOPEN "FILE2",F
 20  FILE 7,F,3  !LOCK FILE
 30  REM PROCESS RECORD
 ....
 90  FILE 8,F  !UNLOCK FILE
```

The FILE 0 and FILE 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 FILE 0 and FILE 1 is valid only until another BASIC command is executed. (This includes another FILE command.) The verb FILE resets the length to 6 bytes. Thus, in order to select a different variable length, a FILE 0 or FILE 1 command must be followed by a semicolon and another file command expression.

```
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
```

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

The sample subroutine to the right illustrates how a software record lock is implemented. Line 8010 selects the file. Lines 8020 through 8040 wait until the file can be locked. Once the task gains exclusive use of the file, line 8050 positions to the desired record.

Line 8060 reads the record lock parameter. If the record is already locked, then the file is unlocked and the whole process is repeated. If the record has been locked, then lines 8080 and 8090 lock the record. Line 8100 reads the record and line 8110 unlocks the file and returns.

```
8000  REM READ & LOCK RECORD <FILEID>
8010  FILE SELECT,FILEID
8020  FILE LOCK,FILEID,ERR
8030  IF ERR=75: GOTO 8020  !FILE LOCKED, TRY AGAIN
8040  IF ERR<>0" GOTO 8500  !ERROR
8050  FILE POSITION,REC_NUM,REC_LEN,0
8060  FILE READF,L
8070  IF L<>0: FILE UNLOCK,FILEID: GOTO 8000
8080  FILE POSITION,REC_NUM,REC_LEN,0
8090  FILE WRITE,-1  !LOCK RECORD
8100  FILE SELECT,FILEID,REC_LEN-6;READF,4[0]
8110  FILE UNLOCK,FILEID
8120  RETURN
```

9.6 BASIC PROGRAM EXAMPLES

9.6.1 BASIC TASK LOCK

A lock task command was not included in BASIC in order to prevent inadvertent system lockups. However, for intertask communications, a lock process command is sometimes necessary. The following illustrates how to lock a task:

```
10  $PLOCK=%:2FC9 045B"  !XLKT > RT
20  $PUNLOCK=%"2FCA 045B"  !XULT > RT
30  CALL #ADR PLOCK  !LOCK TASK
40  FOR I-1 TO 1000: SWAP : NEXT I
50  CALL #ADR PUNLOCK  !UNLOCK TASK
60  FOR I-1 TO 1000: SWAP : NEXT I
70  GOTO 30
```

9.6.2 BASIC SETFILE ATTRIBUTES

File attributes can be changed from BASIC using the following routine:

```
1000  REM SET FILE ATTRIBUTES
1010  COM[0]=ADR N[0]  !POINT TO FILE NAME
1020  COM[1]=ADR A[0]  !POINT TO ATTRIBUTE STRING
1030  COM[2]=%"C0670002C0A7000804C02F8F100004F7C5C0045B"
1040  CALL #ADR COM[2]
1050  IF COM[0]: PRINT "PDOS ERROR";COM[0]
1060  RETURN
```

```
                        *       WRITE ATTRIBUTES
                        *
0000: C067 0002         XWFA    MOV @2(7),R1    ;GET FILE NAME
0004: C0A7 0008                 MOV @8(7),R2    ;GET ATTRIBUTES
0008: 04C0                      CLR R0          ;CLEAR ERROR
000A: 2F8F                      XWFA            ;WRITE ATTRIBUTES
000C: 1000                        NOP
000E: 04F7                      CLR *R7+        ;RETURN ERROR CODE
0010: C6C0                      MOV R0,*R7
00012: 045B                     RT              ;RETURN
0014:         0000'             END XWFA
```

9.6.3 BASIC CREATE TASK

A BASIC task can spawn another task using memory from its own address space. The FREE statement moves down the upper BASIC data structures; namely the EXTERNAL table, FOR/NEXT and GOSUB stacks, and variable storage. In the following example. a command line is passed to the new task in array $L[0]. The task status is monitored by the TSK function. When the spawned task is done, the memory is recovered, again using the FREE command.

```
2000  REM CREATE TASK
2010  DIM CREATE[5],L[10]
2020  FREE 1024  !FREE 1k
2030  $L[0]="LT.LS 10.KT 0"
2040  COM[0]=ADR[L[0]]  !TASK COMMAND LINE
2050  COM[1]=SYS[28]  !LOW MEMORY ADDRESS
2060  COM[2]=SYS[29]  !HIGH MEMORY ADDRESS
2070  COM[3]=1  !TASK TIME
2080  COM[4]=SYS[10]  !CRT PORT
2090  $CREATE[0]=%"05C70700C057 C0A70012C0E7 0018C1270006"
2100  $CREATE[3]=%"C167000C04D7 2FDDC5C0C9C0 0006045B"
2110  CALL #ADR CREATE[0]  !CREATE TASK
2120  IF COM[0]: PRINT "PDOS ERROR";COM[0]: GOTO 2140
2130  IF TSK[COM[1]]>0: GOTO 2120
2140  FREE -1024  !RECOVER SPACE
2150  RETURN
```

```
                    *      IN COM(0) = (TASK COMMAND LINE)
                    *         COM(1) = LOW MEMORY ADDRESS
                    *         COM(2) = HIGH MEMORY ADDRESS
                    *         COM(3) = TASK TIME
                    *         COM(4) = CRT PORT
                    *     OUT COM(0) = ERROR
                    *         COM(1) = RETURNED TASK #
                    *
 0000: 05C7         BTSK  INCT R7          ;MOVE TO PARAMETERS
 0002: 0700               SETO R0          ;USER CURRENT PAGE W/R4,R5
 0004: C057               MOV *R7,R1       ;GET TASK COMMAND LINE POINTER
 0006: C0A7 0012          MOV @3*6(7),R2   ;GET TASK TIME
 000A: C0E7 0018          MOV @4*6(7),R3   ;GET TASK PORT
 000E: C127 0006          MOV @1*6(7),R4   ;GET LOW MEMORY ADDRESS
 0012: C167 000C          MOV @2*6(7),R5   ;GET HIGH MEMORY ADDRESS
 0016: 04D7               CLR *R7          ;CLEAR ERROR RETURN
 0018: 2FDD               XCTB             ;CREATE TASK
 001A: C5C0                 MOV R0,*R7     ;RETURN ERROR
 001C: C9C0 0006          MOV R0,@1*6(7)   ;RETURN TASK NUMBER
 0020: 045B               RT
 0022:      0000'         END BTSK
```

9.6.4 BASIC ARRAY PASSING

Arrays can be passed to functions and subroutines using the EQUATE statement. The data storage address of a dummy variable can be assigned to any memory location. If a dummy variable is assigned to the information vector of an array (array descriptor), then the dummy variable assumes the same dimensions and limits.

Each dimension of an array allocates two words (4 bytes) in the information vector. Thus, a two dimensional array's information vector starts at address ADR[A[0]]-4*2.

```
LIST
 10  DIM A[2,2],B[2,2],C[2,2]
 20  CALL FNFILL[A[0,0],10],FNFILL[B[0,0],10]
 30  CALL FNMUL[A[0,0],B[0,0],C[0,0]]
 40  GOSUB 500
 50  STOP
 500  REM PRINT ARRAYS
 510  PRINT
 520  FOR I=0 TO 2
 530   PRINT #" 990";A[I,0];A[I,1];A[I,2]; TAB 20
 540   PRINT #" 990";B[I,0];B[I,1];B[I,2]; TAB 40
 550   PRINT #" 990";C[I,0];C[I,1];C[I,2]
 560  NEXT I
 570  RETURN

 1000  DEFN FNMUL[I,J,K]
 1010  EQUATE T1[0],ADR[I]-8;T2[0],ADR[J]-8;T3[0],ADR[K]-8
 1020  FOR II=0 TO 2: FOR JJ=0 TO 2
 1030    T3[II,JJ]=T1[2,JJ]*T2[II,0]+T1[1,JJ]*T2[II,1]+T1[0,JJ]*T2[II,2]
 1040   NEXT JJ: NEXT II
 1050  FNEND

 2000  DEFN FNFILL[I,J]
 2010  EQUATE T1[0],ADR[I]-8
 2020  FOR II=0 TO 2: FOR JJ=0 TO 2
 2030    T1[II,JJ]=INT[RND*J]
 2040   NEXT JJ: NEXT II
 2050  FNEND
RUN

   5   3   9           2   8   9          65  43 155
   1   2   7           5   7   3          52  23 121
   6   0   9           7   8   1          55  19 128

STOP AT 50
```

9.6.5 BASIC DISK BACKUP

The following routine copies the disk specified by COM[0] to the disk specified by COM[1], sector by sector.

```
3000  REM DISK BACKUP
3010  BACKUP[8]
3020  COM[0]=0  !SOURCE DISK #
3030  COM[1]=1  !DESTINATION DISK #
3040  COM[2]=1976  !# OF SECTORS
3050  $BACKUP[0]=%"05C7C0D7C127 0006C0E7000C 04D704C1C9C1 0006C003C9C0"
3060  $BACKUP[4]=%"000CC0892FCD 1009C004C9C0 000C2FCE1004 058181411AF0"
3070  $BACKUP[8]=%"045BC5C0045B"
3080  CALL #ADR BACKUP[0]
3090  IF COM[0]: PRINT "PDOS ERR";COM[0];" AT SECTOR";COM[1];" ON DISK";COM[2]
3100  RETURN
```

```
                    *       IN COM(0) = SOURCE DISK UNIT #
                    *          COM(1) = DESTINATION DISK UNIT #
                    *          COM(2) = # OF SECTORS
                    *      OUT COM(0) = ERROR
                    *          COM(1) = # OF SECTORS COPIED
                    *          COM(2) = DISK WITH ERROR
                    *
0000: 05C7          DCPY   INCT R7          ;MOVE TO PARAMETERS
0002: C0D7                 MOV *R7,R3       ;GET SOURCE DISK #
0004: C127 0006            MOV @1*6(7),R4   ;GET DESTINATION DISK #
0008: C0E7 000C            MOV @2*6(7),R3   ;GET # OF SECTORS
000C: 04D7                 CLR *R7          ;CLEAR ERROR RETURN
000E: 04C1                 CLR R1           ;START WITH SECTOR 0
                    *
0010: C9C1 0006     DCPY02 MOV R1,@1*6(7)   ;SET SECTOR # IN COM(1)
0014: C003                 MOV R3,R0        ;GET SOURCE DISK #
0016: C9C0 000C            MOV R0,@2*6(7)   ;SET DISK # IN COM(2)
001A: C089                 MOV R9,R2        ;GET BUFFER POINTER
001C: 2FCD                 XRSE             ;READ SECTOR
001E: 1009                   JMP DCPYE      ;ERROR
0020: C004                 MOV R4,R0        ;GET DESTINATION DISK #
0022: C9C0 000C            MOV R0,@2*6(7)   ;SET DISK # IN COM(2)
0026: 2FCE                 XWSE             ;WRITE SECTOR
0028: 1004                   JMP DCPYE      ;ERROR
002A: 0581                 INC R1           ;NEXT
002C: 8141                 C R1,R5          ;DONE?
002E: 1AF0                   JL DCPY02      ;N
0030: 045B                 RT               ;Y
                    *
0032: C5C0          DCPYE  MOV R0,*R7       ;RETURN ERROR
0034: 045B                 RT               ;RETURN
0036:      0000'           END DCPY
```

9.6.6 FNPOP EXAMPLE

User defined functions must be gracefully exited! Variable addresses and pointers are stored on the system heap and must be restored in an orderly manner. The following example illustrates how the FNPOP command is used to clear the system heap.

```
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
? _
```

9.6.7 BASIC MENUS

Menu driven programs are easily implemented using DATA statements and the MENU subroutine listed below. The menu header also contains the current task number for a multi-user environment. The "/" string in the DATA statements terminates the menu list.

```
LIST

 100  LABEL MAIN
 110  RESTORE 1: GOSUB MENU
 120  DATA "** MASTER MENU"
 130  DATA "VIEW RECORD","ENTER RECORD"
 140  DATA "UPDATE RECORD","QUERY"
 150  DATA "EXIT","/"
 300  ON I: GOSUB VIEW,ENTER,UPDATE,QUERY,QUITS
 320  GOTO MAIN

 1000  LABEL QUITS

 1200  LABEL VIEW

 1400  LABEL UPDATE

 1600  LABEL ENTER
 1610  RETURN

 4000  LABEL QUERY
 4010  RESTORE 1: GOSUB MENU
 4020  DATA "** QUERY MENU"
 4030  DATA "CORRESPONDENCE BY DATE","CORRESPONDENCE BY TYPE"
 4040  DATA "PRODUCTION BY OUTSTANDING BALANCES"
 4050  DATA "PRODUCTION BY OUTSTANDING ORDERS"
 4060  DATA "PRODUCTION BY QUANTITIES & TOTALS"
 4070  DATA "CUSTOMER BY OUTSTANDING LICENSES"
 4080  DATA "MASTER LIST","/"
 4100  RETURN

 9000  LABEL MENU
 9010  DIM T[10]
 9020  READ $T[0]: $I=SYS[36]  !READ HEADING
 9030  $T[0]=$T[0]&" TASK "&$I&" **"  !APPEND TASK NUMBER
 9040  I=0: PRINT @"C";"      ";@[5,24];$T[0]: PRINT
 9050  READ $T[0]: I=I+1: IF $T[0]<>"/"
 9060    THEN PRINT @[I+6,24];#"0) ";I;$T[0]: SKIP -2
 9070  INPUT @[I+7,24];"ENTER SELECTION: ";I
 9080  RETURN
RUN
```

(9.6.7 BASIC MENUS continued)

```
** MASTER MENU TASK 0 **

1) VIEW RECORD
2) ENTER RECORD
3) UPDATE RECORD
4) QUERY
5) EXIT
ENTER SELECTION: 4
```

```
** QUERY MENU TASK 0 **

1) CORRESPONDENCE BY DATE
2) CORRESPONDENCE BY TYPE
3) PRODUCTION BY OUTSTANDING BALANCES
4) PRODUCTION BY OUTSTANDING ORDERS
5) PRODUCTION BY QUANTITIES & TOTALS
6) CUSTOMER BY OUTSTANDING LICENSES
7) MASTER LIST
ENTER SELECTION:
```

```
* MASTER MENU TASK 0 **

1) VIEW RECORD
2) ENTER RECORD
3) UPDATE RECORD
4) QUERY
5) EXIT
ENTER SELECTION:_
```

9.6.8 BASIC STATUS LINE PROCESSOR

The 24th line of a terminal can be used to display status and prompt for user inputs. Many different types of entries can be accepted there including a command to dump the screen to a printer.

Subroutine FNSTATUS_LINE returns a line number to the calling program according to a single character input. The first parameter, a string, is printed on the 24th line. The second and third parameters are the line number values returned on a non 'Y' and 'Y' input, respectively. If a control 'P' is entered and event 61 is 0, a copy of the screen is output to UNIT 2 at 9600 baud.

For the screen dump to work, the terminal must be capable of reading the display screen and sending a line at a time under software control back to the computer. This example uses the escape sequence of '<esc>4' to send the current line. These control characters are the first two characters of element PL[4].

```
2000  LABEL ENTER
2010  GOSUB INPUT_MASTER_RECORD
2020  GOSUB PRINT_MASTER_RECORD
2030  GOTO FNSTATUS_LINE["ENTRY OK? N",2080,2070]
2040  GOSUB WRITE_MASTER_REC
2050  RETURN

7200  DEFN FNSTATUS_LINE[S,CR,Y]
7210  PRINT @[23,0]; TAB 40;@[23,0];$S;"<08>";
7220  INPUT ?INERR;#1;$I;
7230  IF $I="Y": FNSTATUS_LINE=Y: FNEND
7240  FNSTATUS_LINE=CR
7250  FNEND

7300  LABEL INERR
7310  IF SYS[0]<>16: PRINT "<07>";: RETURN -2
7320  EVENT 61,I: IF I=0: GOTO DUMP_SCREEN
7330  PRINT @[23,0]; TAB 30;@[23,0];"PRINTER BUSY.  PLEASE WAIT!";
7340  SWAP : IF KEY[0]=0: GOTO 7320
7350  RETURN -2
```

(9.6.8 BASIC STATUS LINE PROCESSOR continued)

```
7400  LABEL DUMP_SCREEN
7410  DATE $L[0]: TIME $L[2]: BAUD -2,1: UNIT 2
7420  PRINT "<0C>**** TASK =";MEM[02FE5H];" *** DATE = ";$L[0];
7430  PRINT " *** TIME = ";$L[2];" ***************************": PRINT
7440  $PL[0]=%"C14B2E06C2E5 0002069BC1C2 C0460202004F 2F5D2FC90200"
7450  $PL[4]=%"1B342F582F56 DDC002800000 16FB2FCA0607 75D70455"
7460  FOR I=0 TO 22
7470   UNIT 1: CALL #ADR PL[0],I,L[0]
7480   UNIT 2: PRINT $L[0]
7490  NEXT I
7500  PRINT : FOR I=1 TO 78: PRINT "*";: NEXT I: PRINT
7510  UNIT 1: EVENT -61  !RELEASE PRINTER
7520  RETURN -2
```

```
                           *      DUMPS:SR        05/05/82
                           *
       2E00                       DXOP EVFIX,8    ;EVALUATE INTEGER
                           *
                           *      CALL #GETL,ROW,A(0)
                           *
 0000: C14B                DMPS   MOV R11,R5      ;SAVE RETURN
 0002: 2E06                       EVFIX R6        ;GET ROW
 0004: C2E5 0002                  MOV @2(5),R11
 0008: 069B                       BL *R11         ;GET EVAL
 000A: C1C2                       MOV R2,R7       ;SAVE ARRAY ADDRESS
                           *
 000C: C046                DMPS02 MOV R6,R1       ;SET ROW
 000E: 0202 004F                  LI R2,79        ;COLUMN=79
 0012: 2F5D                       XPSC            ;POSITION CURSOR
 0014: 2FC9                       XLKT            ;LOCK TASK
 0016: 0200 1B34                  LI R0,>1B00+'4'
 001A: 2F58                       XPCC            ;SEND READ LINE COMMAND
                           *
 001C: 2F56                DMPS04 XGCR            ;GET CHARACTER
 001E: DDC0                       MOVB R0,*R7+    ;STORE
 0020: 0280 0000                  CI R0,>0D00     ;CR?
 0024: 16FB                         JNE DMPS04    ;N
 0026: 2FCA                       XULT            ;Y, UNLOCK TASK
 0028: 0607                       DEC R7          ;BACKUP
 002A: 75D7                       SB *R7,*R7      ;NULL STRING
 002C: 0455                       B *R5
 002E:      0000'                 END DMPS
```

9.6.9 BASIC INPUTS AND PROMPTS

Line prompts, cursor positions, and other control functions can be passed as arguments to generalized input functions. This gives flexibility to data verification and movement through menu input sequences.

The functions FUNSTRING, FNPHONE, FNNUMBER, and FNDATE illustrate how to use BASIC functions to easily build input menus. The value of each function is used as a line number for a GOTO statement. This allows the program to move back to the last prompt to correct or input new data.

The first two parameters of each function are the X and Y cursor position for the prompt string, which is the third parameter. The fourth parameter is the array entry number or variable where the input is stored. The fifth parameter is either the string length or an echo mask. The last two parameters are the control line numbers. If a control B is entered, the last parameter is returned. Otherwise, the second to the last parameter is returned as the function value.

The following example illustrates the four different types of input functions. The IFLAG variable is set to 1 if an input occurred.

```
10 DIM R[4,5],T[10]
20 GOSUB INPUT_MASTER_RECORD
30 STOP

1000 LABEL INPUT_MASTER_RECORD
1010 GOTO FNSTRING[3,10,"ENTER NAME: ",0,24,1020,1050]
1020 GOTO FNPHONE[4,9,"ENTER PHONE: ",1,1030,1010]
1030 GOTO FNNUMBER[5,8,"ENTER NUMBER: ",N,"<<<,<<0.00>",1040,1020]
1040 GOTO FNDATE[6,10,"ENTER DATE: ",2,1050,1030]
1050 RETURN
```

(9.6.9 BASIC INPUTS AND PROMPTS continued)

```
2000 DEFN FNSTRING[R,C,S,I,L,N,B]
2010 LOCAL II
2020 PRINT @[R,C];$S;
2030 FOR II=1 TO L: PRINT "_";: NEXT II: II=C+LEN S
2040 INPUT ?2100;@[R,II];#L;$T[0];: IF $T[0]="\",1: $T[0]="": SKIP 1
2050 IF $T[0]<>""
2060   THEN $R[I,0]=$T[0]: IFLAG=1
2070   THEN PRINT @[R,II]; TAB C+L+1+LEN S;@[R,II];$T[0];
2080   ELSE IF UFLAG=0: $R[I,0]=""
2090 FNSTRING=N: FNEND
2100 POP : IF SYS[0]=2: FNSTRING=B: FNEND
2110 PRINT "<07>";: GOTO 2020
2120 FNEND

2200 DEFN FNPHONE[R,C,S,I,N,B]
2210 PRINT @[R,C];$S;"(___) ___-____ ext ____";@[R,C+1+LEN S];
2220 INPUT ?2300;#3;I1;: IF I1=0: FNPHONE=N: FNEND
2230 IFLAG=1: INPUT ?2300;@[R,C+6+LEN S];%3;I2;%4;"-";I3;" ext ";#4;I4;
2240 $R[I,0]=#"(000) 000-0000",I1*10000000+I2*10000+I3
2250 $R[I,0]=$R[I,0]&#" ext 0000",I4: PRINT @[R,C+LEN S];$R[I,0];
2260 FNPHONE=N: FNEND
2300 POP : IF SYS[0]=2: FNPHONE=B: FNEND
2310 PRINT "<07>";: GOTO 2210
2320 FNEND

2400 DEFN FNNUMBER[R,C,S1,I,S2,N,B]
2410 LOCAL II,E,L
2420 PRINT @[R,C];$S1;: L=LEN S2-1
2430 FOR II=1 TO L: PRINT "_";: NEXT II: II=C+LEN S1
2440 INPUT ?2500;@[R,II];#L;$T[0];
2450 IF $T[0]<>"": I=$T[0],E: IF E<>"": PRINT "<07>";: GOTO 2420
2460 PRINT @[R,II]; TAB C+L+II;@[R,II];#$S2;I;
2470 IFLAG=1: FNNUMBER=N: FNEND
2500 POP : IF SYS[0]=2: FNNUMBER=B: FNEND
2510 PRINT "<07>";: GOTO 2420
2520 FNEND

2600 DEFN FNDATE[R,C,S,I,N,B]
2610 LOCAL I1,I2,I3,II
2620 PRINT @[R,C];$S;"mm/dd/yy";: II=C+LEN S
2630 INPUT ?2700;@[R,II];#2;I1;: IF I1=0: FNDATE=N: FNEND
2640 IFLAG=1: INPUT ?2700;@[R,II+2];"/";#2;I2;
2650 INPUT ?2700;@[R,II+5];#2;"/";I3;
2660 $R[I,0]=#"00/00/00",I1*10000+I2*100+I3: PRINT @[R,II];$R[I,0];
2670 FNDATE=N: FNEND
2700 POP : IF SYS[0]=2: FNDATE=B: FNEND
2710 PRINT "<07>";: GOTO 2620
2720 FNEND
```

9.6.10 ASSIGN CONSOLE INPUTS

The SYS[12] variable specifies that all further keyboard inputs are to come from a file specified by <exp>. The file must be opened before the SYS[12] assignment is made. Any error in the input file (specifically an END-OF-FILE) closes the file and reverts back to the user keyboard for input.

If SYS[12] is equal to zero, then further character inputs again come from the keyboard. This is used to switch temporarily between the keyboard and a file for inputs.

```
DISPLAY "INDATA"
GEORGE RICHARDS
1455 NORTHWOOD AVE
DEAN C. CAMPBELL
1004 EAST WEDGEFIELD
JOHN HEMPS
254 UNIVERSITY AVE
LIST
 10  DIM NAME[10],ADDRESS[10]
 20  OPEN "INDATA",FILEID
 30  SYS[12]=FILEID
 100  INPUT "NAME=";$NAME[0]
 120  INPUT "ADDRESS=";$ADDRESS[0]
 130  SYS[12]=0!  REVERT TO KEYBOARD
 140  INPUT "OK?";$I
 150  IF $I="Y",1: GOTO 30
 160  STOP
RUN
NAME=GEORGE RICHARDS
ADDRESS=1455 NORTHWOOD AVE
OK?Y
NAME=DEAN C. CAMPBELL
ADDRESS=1004 EAST WEDGEFIELD
OK?Y
NAME=JOHN HEMPS
ADDRESS=254 UNIVERSITY AVE
OK?Y
NAME=_
```

9.7 BASIC PROGRAMMING TIPS

Generally, good, efficient code results from a good understanding of the language. There seems to always be a better way of implementing any algorithm and hence the purpose of this section is to acquaint you with some subtle but useful programming tips.

1. Integer or byte data storage for file data compaction.

```
MEM[ADR A]=25
MEM[ADR A+1]=100
MEMW[ADR A+2]-2048
MEMW[ADR A+4]=-30000
BINARY 1,F;2,A
```

2. Get BASIC memory limits for maximum array size. SYS[24] and SYS]25] are the memory bounds for the free or available memory space of PDOS BASIC. A simple program allows dynamic array allocation for maximum array size.

```
1000  REM *** CALCULATE MEMORY BOUNDS ***
1010  I=24  !BYTES/ARRAY ELEMENT
1020  I0=SYS[24]: IF I0<0: I0=2^16+I0
1030  I1=SYS[25]: IF I1<0: I1=2^16+I1
1040  L=INP[(I1-I0-200)/I]  !200=OVERHEAD
1050  DIM ARRAY [L,3]
```

3. Set random seed from system clocks. The random seed is a 16-bit number from which all BASIC random numbers are generated. There are several system clock parameters available to set the seed.

```
LIST
 10  PRINT MEMW[02F88H],MEMW[02FE2H],MEM[02F89H]*MEM[02FE1H]
 20  GOTO 10
RUN
 9572          99          1272
 9580          107         1368
 9588          115         1464
 9596          123         1560
 9604          9           1692
 9615          17          1788
 9623          25          1884
 9631          25          1980
 9639          41          2076

ESCAPE AT 20
```

(9.7 BASIC PROGRAMMING TIPS continued)

4. Timing routines. The delta TIC function can easily be used to time portions of an BASIC program.

```
LIST
 10  T=TIC 0
 20  FOR I=1 TO 1000
 30   J=SIN I
 40  NEXT I
 50  PRINT "ELAPSED TIME=";TIC[T]/125;"SECONDS."
RUN
ELAPSED TIME= 17.384 SECONDS

STOP AT 50
```

5. BASIC program protection. A BASIC program can be protected from being listed by the following method:

1. First line is 'NOESC'.
2. RENUMB program with negative line numbers.
3. SAVEB program.

Such a program can be neither interrupted with an <esc> nor listed with the LIST or LISTRP commands.