MOSTEK PROCESS BASIC
Version 0.A
Reference Manual

Published by MOSTEK Corporation with the permission of MICROSOFT.

Publication No. MK79874

TABLE OF CONTENTS

PARAGRAPH NUMBER	TITLE	PAGE NUMBER
1	GENERAL INFORMATION ABOUT PBASIC	
1.1	INITIALIZATION	1-1
1.2	MODES OF OPERATION	1-2
1.3	LINE FORMAT	1-2
1.3.1	LINE NUMBERS	1-2
1.4	CHARACTER SET	1-2
1.4.1	Control Characters	1-3
1.5	CONSTANTS	1-4
1.5.1	SINGLE AND DOUBLE PRECISION NUMERIC CONSTANTS	1-5
1.6	VARIABLES	1-6
1.6.1	VARIABLE NAMES AND DECLARATION CHARACTERS	1-6
1.6.2	ARRAY VARIABLES	1-7
1.7	TYPE CONVERSION	1-7
1.8	EXPRESSIONS AND OPERATORS	1-8
1.8.1	Arithmetic Operators	1-8
1.8.1.1	Integer Division And Modulus Arithmetic	1-9
1.8.1.2	Overflow And Division By Zero	1-10
1.8.2	Relational Operators	1-10
1.8.3	Logical Operators	1-11
1.8.4	Functional Operators	1-13
1.8.5	String Operations	1-13
1.8.6	OPERATOR PRECEDENCE	1-14
1.9	INPUT EDITING	1-15
1.10	ERROR MESSAGES	1-15
2	PBASIC COMMANDS AND STATEMENTS	
2.1	AUTO	2-2
2.2	CALL	2-3
2.3	CLEAR	2-4
2.4	CONT	2-5
2.5	DATA	2-6
2.6	DEF FN	2-7
2.7	DEFINT/SNG/DBL/STR	2-8
2.8	DEF USR	2-9
2.9	DELETE	2-10
2.10	DIM	2-11
2.11	EDIT	2-12
2.12	END	2-16
2.13	ERASE	2-17
2.14	ERR AND ERL VARIABLES	2-18
2.15	ERROR	2-19
2.16	FOR...NEXT	2-21
2.17	GETCLK	2-23
2.18	GOSUB...RETURN	2-24
2.19	GOTO	2-25
2.20	IF...THEN[...ELSE]	2-26
2.21	IF...GOTO	2-26
2.22	INPUT	2-28
2.23	LET	2-30
2.24	LINE INPUT	2-31
2.25	LIST	2-32
2.26	LLIST	2-34

TABLE OF CONTENTS

PARAGRAPH NUMBER	TITLE	PAGE NUMBER
2.27	LPRINT AND LPRINT USING	2-35
2.28	MID$	2-36
2.29	NEW	2-37
2.30	NULL	2-38
2.31	ON ERROR GOTO	2-39
2.32	ON...GOSUB	2-40
2.33	ON...GOTO	2-40
2.34	OPTION BASE	2-41
2.35	OUT	2-42
2.36	POKE	2-43
2.37	PPROM	2-44
2.38	PRINT	2-47
2.39	PRINT USING	2-49
2.40	RANDOMIZE	2-53
2.41	READ	2-54
2.42	REM	2-56
2.43	RENUM	2-57
2.44	RESTORE	2-58
2.45	RESUME	2-59
2.46	RPROM	2-60
2.47	RUN	2-63
2.48	SETCLK	2-64
2.49	STOP	2-65
2.50	SWAP	2-66
2.51	TRON/TROFF	2-67
2.52	WAIT	2-68
2.53	WHILE...WEND	2-69
2.54	WIDTH	2-70
2.55	WRITE	2-71
3	PBASIC FUNCTIONS	
3.1	ABS	3-2
3.2	ASC	3-3
3.3	ATN	3-4
3.4	BCD	3-5
3.5	BIN	3-6
3.6	CDBL	3-7
3.7	CHR$	3-8
3.8	CINT	3-9
3.9	COS	3-10
3.10	CSNG	3-11
3.11	DATE$	3-12
3.12	EXP	3-13
3.13	FIX	3-14
3.14	FRE	3-15
3.15	HEX$	3-16
3.16	INP	3-17
3.17	INPUT$	3-18
3.18	INSTR	3-19
3.19	INT	3-20
3.20	LEFT$	3-21
3.21	LEN	3-22
3.22	LOG	3-23

TABLE OF CONTENTS

PARAGRAPH NUMBER	TITLE	PAGE NUMBER
3.23	LPOS	3-24
3.24	MID$	3-25
3.25	OCT$	3-26
3.26	PEEK	3-27
3.27	POS	3-28
3.28	RIGHT$	3-29
3.29	RND	3-30
3.30	ROTATE	3-31
3.31	SGN	3-32
3.32	SHIFT	3-33
3.33	SIN	3-34
3.34	SPACE$	3-35
3.35	SPC	3-36
3.36	SQR	3-37
3.37	STR$	3-38
3.38	STRING$	3-39
3.39	TAB	3-40
3.40	TAN	3-41
3.41	TIME$	3-42
3.42	USR	3-43
3.43	VAL	3-44
3.44	VARPTR	3-45
A	SYSTEM CONFIGURATION	
A.1	MEMORY LAYOUT	A-1
A.1.1	PBASIC INTERPRETER	A-1
A.1.2	PBASIC APPLICATION PROGRAM	A-1
A.2	HARDWARE CONFIGURATION	A-3
A.3	BOARD LEVEL DOCUMENTATION	A-3
B	SYSTEMS GENERATION	
B.1	PBASIC INTERPRETER	B-1
B.1.1	INITIAL PROGRAM DEVELOPMENT	B-1
B.1.2	PROMMING APPLICATION PROGRAM	B-1
B.1.3	RUNNING PROM APPLICATION IN DEVELOPMENT CONFIGURATION	B-1
B.1.4	READING PROMMED APPLICATION INTO RAM	B-1
B.1.5	AUTOMATIC RUNTIME MODE	B-2
C	INTEGRITY TEST	
C.1	INTEGRITY TEST PURPOSE	C-1
C.2	INTEGRITY TEST SCHEME	C-1
D	MATHEMATICAL FUNCTIONS	
D.1	PBASIC MATH VERSIONS	D-1
D.1.1	PBASIC SOFTWARE (S/W) MATH VERSION (PBASIC)	D-1
D.1.2	PBASIC HARDWARE (H/W) 9511 MATH VERSION (PBASIC/9511)	D-1
D.2	DERIVED FUNCTIONS	D-1
E	Converting Programs to PBASIC	
E.1	STRING DIMENSIONS	E-1
E.2	MULTIPLE ASSIGNMENTS	E-1
E.3	MULTIPLE STATEMENTS	E-2
E.4	MAT FUNCTIONS	E-2

TABLE OF CONTENTS

PARAGRAPH NUMBER	TITLE	PAGE NUMBER
F	Summary of Error Codes and Error Messages	
G	Assembly language Subroutines	
G.1	MEMORY ALLOCATION	G-1
G.2	USR FUNCTION CALLS	G-1
G.3	CALL STATEMENT	G-2
G.4	INTERRUPTS	G-4
H	ASCII Character Codes	

INTRODUCTION

MOSTEK PROCESS BASIC, or PBASIC, is the most extensive implementation of a Process Control BASIC available for the Z80 microprocessor. PBASIC provides the process control user with a non-volatile, ROMable, monitor which in turn has the added capability of placing the application program into PROM. This provides for a completely non-volatile system that can be executed by simply turning on the power. PBASIC is a stand-alone executive and requires no operating system interface to execute.

MOSTEK PBASIC is an extended implementation of Microsoft BASIC-80 version 5.0 for the Z80 microprocessor, and is among the fastest microprocessor BASICs available. PBASIC is implemented as an interpreter and is highly suitable for user interactive processing. Programs and data are stored in a compressed internal format to maximize memory utilization. In a 64k system, up to 28k bytes of memory are available for the user's program and data.

This manual is divided into three chapters plus a number of appendices. Chapter 1 covers the representation of information utilized by PBASIC. Chapter 2 contains the syntax and semantics of every command and statement in PBASIC, ordered alphabetically. Chapter 3 describes all of PBASIC's intrinsic functions, also ordered alphabetically. The appendices contain information on system configuration, system generation, math functions, and integrity test; plus lists of error messages, ASCII codes, and helpful information on assembly language subroutines.

MOSTEK

Z80 MICROCOMPUTER SYSTEMS

PROCESS BASIC SOFTWARE INTERPRETER

FEATURES

- Extensive Process Control Features
- Occupies 24k bytes and can execute out of PROMs
- Program and Read Application in/out of PROMs
- Optional High-Speed Math Version utilizing MDX-Math card
- Stand-alone operation, i.e. requires no operating system
- Insures program integrity
- Debug or Runtime execution on power-up
- Direct access to CPU I/O Ports
- Ability to read or write any memory location (PEEK, POKE)
- Arrays with up to 255 dimensions
- Dynamic allocation and de-allocation of arrays
- IF...THEN...ELSE and IF...GOTO (both if's may be nested)
- WHILE...WEND structured construct
- Direct (immediate) execution of statements
- Error trapping, with error messages in English
- Four variable types: Integer, string, single and double precision real
- Long variable names significant up to 40 characters
- Full PRINT USING capabilities for formatted output
- Extensive program editing facilities
- Trace facilities
- Ability to call any number of assembly language subroutines
- Supports console and line printer I/O

LANGUAGE COMMANDS SUMMARY

Commands:

AUTO	CLEAR	CONT	DELETE	EDIT
LIST	LLIST	NEW	NULL	PPROM
RENUM	RPROM	RUN	TROFF	TRON
WIDTH				

Program Statements:

CALL	DEFDBL	DEF FN	DEFINT	DEFSNG
DEFSTR	DEFUSR	DIM	END	ERASE
ERROR	FOR...NEXT	GETCLK	GOSUB...RETURN	GOTO
IF...THEN(ELSE)	IF...GOTO	LET	ON ERROR GOTO	ON...GOSUB
ON...GOTO	OPTION BASE	POKE	RANDOMIZE	REM
RESUME	SETCLK	STOP	SWAP	WAIT
WHILE...END				

Input/Output Statements:

DATA	INPUT	LINE INPUT	LPRINT	LPRINT USING
OUT	PRINT	PRINT USING	READ	RESTORE
WRITE				

Operators:

=	-	+	*	/
^	\	>	<	<=
>=	<>	MOD	NOT	AND
OR	XOR	IMP	EQU	

Arithmetic Functions:

ABS	ATN	BCD	BIN	CDBL
CINT	COS	CSNG	ERL	ERR
EXP	FIX	FRE	INT	LOG
PEEK	RND	ROTATE	SGN	SHIFT
SIN	SQR	TAN	USR	VARPTR

String Functions:

ASC	CHR$	DATE$	HEX$	INSTR
LEFT$	LEN	MID$	OCT$	RIGHT$
SPACE$	STR$	STRING$	TIME$	VAL

Input/Output Functions:

INP	INPUT$	LPOS	POS	SPC
TAB				

CHAPTER 1

GENERAL INFORMATION ABOUT PBASIC

1.1 INITIALIZATION

At system power-up, PBASIC checks to see if application source RAM or a valid PROM application is present. If neither of these are resident, then the following unrecoverable error message is printed:

```
***** NO RAM MEMORY - OR - BAD FIRST CHIP *****
```

The user must supply RAM or a valid PROMed application for PBASIC to execute.

PBASIC can start execution in either a Debug or Runtime configuration. In the Debug configuration the system initializes for operator interface. In the Runtime configuration the system initializes to start executing the application program that currently resides in the system. The selection of Debug/Runtime configuration is determined by hardware (H/W) strapping 0000H/E000H (HEX) execution address respectively.

On Debug system power-up, PBASIC prints the appropriate sign-on message to the console for either the software Math version as follows:

```
PBASIC  REV  1.0
MOSTEK PROCESS CONTROL BASIC
COPYRIGHT 1980 (C) BY MOSTEK CORP.
```

or for the hardware 9511 Math version:

```
PBASIC/9511  REV  1.0
MOSTEK PROCESS CONTROL BASIC W/9511
COPYRIGHT 1980 (C) BY MOSTEK CORP.
```

The difference in the two versions is the software Math version simulates all math functions via software routines and the hardware Math version requires an AM9511 math processor board (MDX-MATH). If the AM9511 is not present, when executing the hardware Math version, the error message

```
<><><> AM9511 H/W NOT PRESENT <><><>
```

is output to the terminal and all math operations will result in an "Overflow" error condition. The MDX-MATH must be present for the PBASIC/9511 version to operate properly.

The sign-on message is followed by the number of free bytes which represent the amount of RAM space available for PBASIC program and/or string variable storage. The user may now enter PBASIC commands or statements.

On Runtime power-up, PBASIC does not print the sign-on message and starts immediate execution of the application program currently residing in non-volatile memory.

1.2 MODES OF OPERATION

When PBASIC is initialized, it types the prompt "Ok". This means PBASIC is at command level, that is, it is ready to accept commands. At this point, PBASIC may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, PBASIC statements and commands are not preceded by line numbers, and are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use. The last entered instruction may be retrieved by typing Control-A, but previously entered lines are not saved. This mode is useful for debugging and for using PBASIC as a "calculator" for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

1.3 LINE FORMAT

Program lines in a PBASIC program have the following format (square brackets [] indicate optional user specified data):

nnnnn PBASIC statement[:PBASIC statement...] <carriage return>

At the programmer's option, more than one PBASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A PBASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of 255 characters.

In PBASIC, it is possible to extend a logical line over more than one physical line by use of the terminal's <line feed> key. <Line feed> lets you continue typing a logical line on the next physical line without entering a <carriage return>.

1.3.1 LINE NUMBERS

Every PBASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529. A period (.) may be used in EDIT, LIST, LLIST, AUTO and DELETE commands to refer to the current line.

1.4 CHARACTER SET

The PBASIC character set is comprised of alphabetic characters, numeric characters and special characters.

The alphabetic characters in PBASIC are the upper case and lower case letters of the alphabet. All lower case characters are converted to upper case, except those within a string constant or a remark.

The numeric characters in PBASIC are the digits 0 through 9.

The following special characters and terminal keys are recognized by PBASIC:

Character	Name
	Blank
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number or pound sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
<rubout>	Deletes last character typed.
<escape>	Escapes Edit Mode subcommands. See Section 2.11
<tab>	Moves print position to next tab stop. Tab stops are every eight columns.
<line feed>	Moves to next physical line without terminating current line.
<carriage return>	Terminates input of a line.

1.4.1 Control Characters

The following control characters are recognized by PBASIC:

Control-A Enters Edit Mode on the line typed or being typed.

Control-C Interrupts program execution and returns to PBASIC command level. The direct command CONT

resumes execution if the program has not been modified.

Control-G Rings the bell at the terminal.

Control-H Backspace. Deletes the last character typed.

Control-I Tab. Tab stops are every eight columns.

Control-O Halts program output while execution continues. A second Control-O restarts output to the terminal.

Control-R Retypes the line that is currently being typed.

Control-S Suspends program execution. Any subsequent character resumes execution.

Control-U Deletes the line that is currently being typed.

1.5 CONSTANTS

Constants are the actual values PBASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Double quotation characters can only be placed into a string expression with the CHR$ and STRING$ functions, see Sections 3.7 or 3.38 respectively. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
"STRING"
```

Numeric constants are positive or negative numbers. Numeric constants in PBASIC cannot contain commas. There are five types of numeric constants:

1. Integer Constants

Whole numbers between -32768 and +32767. Integer constants do not have decimal points.

2. Fixed Point Constants

Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating Point Constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optional signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). For

the range of single and double precision values, see Appendix D.

Examples:

```
235.988E-7 = .0000235988
2359E6 = 2359000000
```

(Double precision floating point constants use the letter D instead of E. See Section 1.5.1)

4. Hex Constants

Hexadecimal numbers with the prefix &H.

Examples:

```
&H76
&H32F
```

5. Octal Constants

Octal numbers with the prefix &O or &.

Examples:

```
&O347
&1234
```

1.5.1 SINGLE AND DOUBLE PRECISION NUMERIC CONSTANTS

Numeric constants may be either single precision or double precision numbers. With double precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits.

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or
2. exponential form using E, or
3. a trailing exclamation point (!)

A double precision constant is any numeric constant that has:

1. eight or more digits, or
2. exponential form using D, or
3. a trailing number sign (#)

Examples:

Single Precision Constants	Double Precision Constants

46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

1.6 VARIABLES

Variables are names used to represent values that are used in a PBASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

1.6.1 VARIABLE NAMES AND DECLARATION CHARACTERS

PBASIC variable names may be up to 40 characters in length. The characters allowed in a variable name are letters, numbers and the decimal point. The first character must be a letter. Special type declaration characters are also allowed -- see below.

A variable name may not be a reserved word, however embedded reserved words are allowed. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all PBASIC commands, statements, function names and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single or double precision values. The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a numeric variable name is single precision.

Examples of PBASIC variable names follow.

PI#	declares a double precision value
MINIMUM!	declares a single precision value
LIMIT%	declares an integer value
N$	declares a string value
ABC	represents a single precision value

There is a second method to declare variable types. PBASIC statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in the

program to declare the types for certain variable names. These statements are described in detail in Section 2.7.

1.6.2 ARRAY VARIABLES

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with integers or integer expressions. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimensional array, T(1,4) would reference a value in a two-dimensional array, and so on.

1.7 TYPE CONVERSION

When necessary, PBASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)

Example:

```
10 A%= 23.42
20 PRINT A%
RUN
 23
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

Examples:

```
10 D# = 6#/7
20 PRINT D#
RUN
 .8571428571428571
```

The arithmetic was performed in double precision and the result was returned to D# as a double precision value.

```
10 D = 6#/7
20 PRINT D
RUN
 .857143
```

The arithmetic was performed in double precision and the result was returned to D (single precision variable), rounded and printed as a single precision value.

3. Logical operators (see Section 1.8.3) convert their operands to integers and return an integer result. Operands must be in the range

-32768 to 32767 or an "Overflow" error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded.

Example:

```
10 C%= 55.88
20 PRINT C%
RUN
 56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 5.97E-8 times the original single precision value.

Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04  2.039999961853027
```

1.8 EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by PBASIC are divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

1.8.1 Arithmetic Operators

The arithmetic operators in order of precedence are:

Operator	Operation	Sample Expression

^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating Point Division	X*Y X/Y
\	Integer Division	X\Y
MOD	Modulus Arithmetic	X MOD Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained. Here are some sample algebraic expressions and their PBASIC counterparts.

Algebraic Expression	PBASIC Expression
X+2Y	X+Y*2
$X - \frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
$X^{(Y^Z)}$	X^(Y^Z)
X(-Y)	X*(-Y) Two consecutive operators must be separated by parentheses.

1.8.1.1 Integer Division And Modulus Arithmetic

Two additional operators are available: integer division and modulus

arithmetic.

Integer division is denoted by the backslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer. For example:

```
10\4 = 2
25.68\6.99 = 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2 (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

1.8.1.2 Overflow And Division By Zero

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

1.8.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow. (See IF, Section 2.20.)

Operator	Relation Tested	Expression
=	Equality	X=Y
<>	Inequality	X<>Y

<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET, Section 2.23) When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

1.8.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1

0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF, Section 2.20). For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
IF FOPEN THEN 200
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)

-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

1.8.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. PBASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of PBASIC's intrinsic functions are described in Chapter 3. PBASIC also allows "user defined" functions that are written by the programmer. See DEF FN, Section 2.6.

1.8.5 String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=    <>    <    >    <=    >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"      where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison

expressions must be enclosed in quotation marks.

1.8.6 OPERATOR PRECEDENCE

The numeric operators in order of precedence are:

Operator	Operation	Sample Expression
	intrinsic function	SQR(X)
^	exponentiation	X^Y
-	negation	-X
*,/	Multiplication, floating point division	X*Y
\	integer division	X\Y
MOD	modulus	X MOD Y
+,-	addition, subtraction	X+Y
=,<>,>,<, <=,>=	equal, not equal, greater,less,less than or equal, greater than or equal	X>Y
NOT	ones complement	NOT X
AND	logical conjunctive	X AND Y
OR	inclusive or	X OR Y
XOR	exclusive or	X XOR Y
IMP	implication	X IMP Y
EQV	equivalence	X EQV Y

The string operators in order of precedence are:

Operator	Operation	Sample expression
	intrinsic function	LEFT$ ("ABC",2)
+	concatenation	"ABC" + "DEF"
=,<>,<,>, <=,>=	equal, not equal, less, greater, less than or equal, greater than or equal	X$ > "AB"

1.9 INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the RUBOUT key or with Control-H. Rubout surrounds the deleted character(s) with backslashes, and Control-H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type Control-U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. PBASIC will automatically replace the old line with the new line.

More sophisticated editing capabilities are provided in the EDIT command, see Section 2.11.

To delete the entire program that is currently residing in memory, enter the NEW command. (See Section 2.29) NEW is usually used to clear memory prior to entering a new program.

1.10 ERROR MESSAGES

If PBASIC detects an error that causes program execution to terminate, an error message is printed. For a complete list of PBASIC error codes and error messages see Appendix F.

CHAPTER 2

PBASIC COMMANDS AND STATEMENTS

All of the PBASIC commands and statements are described in this chapter. Each description is formatted as follows:

FORMAT:

Shows the correct format for the instruction. See below for format notation.

PURPOSE:

Tells what the instruction is used for.

REMARKS:

Describes in detail how the instruction is used.

EXAMPLE:

Shows sample programs or program segments that demonstrate the use of the instruction.

NOTE:

Any additional information pertinent to that instruction.

Format Notation

Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.

2. Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.

3. Items in square brackets ([ ]) are optional.

4. The slash (/) character indicates alternation: a choice between two or more items.

5. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.

6. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).

2.1 AUTO

FORMAT:

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

PURPOSE:

To generate a line number automatically after every carriage return.

REMARKS:

AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing Control-C. The line in which Control-C is typed is not saved. After Control-C is typed, PBASIC returns to command level.

NOTE:

This command is not allowed within the source program when executing PPROM (see Section 2.37).

EXAMPLE:

```
AUTO              Generates the following:
10 ...
20 ...
30 ...
40 ...
 .
 .

AUTO 100,50       Generates the following:
100 ...
150 ...
200 ...
250 ...
 .
 .
```

2.2 CALL

FORMAT:

CALL <variable name>[(<argument list>)]

PURPOSE:

To call an assembly language subroutine.

REMARKS:

The CALL statement is one way to transfer program flow to an assembly language subroutine. (Also see the USR function, Section 3.42) <variable name> is a numeric variable that contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the assembly language subroutine. See Appendix G for a discussion of the linkage between PBASIC and assembly language routines.

EXAMPLE:

```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
 .
 .
 .
```

2.3 CLEAR

FORMAT:

CLEAR

PURPOSE:

To set all numeric variables to zero and all string variables to null.

EXAMPLE:

```
10 A$="STRING":B=10
20 PRINT A$,B
30 CLEAR
40 PRINT A$,B
RUN
STRING          10
                0
Ok
```

2.4 CONT

FORMAT:

CONT

PURPOSE:

To continue program execution after a Control-C has been typed, or a STOP or END statement has been executed.

REMARKS:

Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number.

NOTE:

CONT is invalid if the program has been modified during the break.

EXAMPLE:

See example Section 2.49, STOP.

2.5 DATA

FORMAT:

DATA <list of constants>

PURPOSE:

To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Section 2.41)

REMARKS:

DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning or starting at a specified line number by use of the RESTORE statement (Section 2.44).

EXAMPLE:

See examples in Section 2.41, READ.

2.6 DEF FN

FORMAT:

DEF FN<name>[(<parameter list>)]=<function definition>

PURPOSE:

To define and name a function that is written by the user.

REMARKS:

<name> must be a legal numeric or string variable name. This name, prefixed with FN, becomes the name of the function. <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call. User-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs. If an error is made in the function definition, a "Syntax Error" condition will occur upon the first execution of the statement referring to that function.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

EXAMPLE:

```
        .
        .
      410 DEF FNAB(X,Y)=X^3/Y^2
      420 T=FNAB(I,J)
        .
        .
```

Line 410 defines the function FNAB. The function is called in line 420.

2.7 DEFINT/SNG/DBL/STR

FORMAT:

DEF<type> <range of letters>

where:
<type> is INT, SNG, DBL or STR and
<range of letters> is a list of one or more items. Each item is either a single letter or is two letters separated by a dash. If the latter is specified, the alphabetic ordering of the second letter must not come before that of the first letter.

PURPOSE:

To declare variable types as integer, single precision, double precision, or string.

REMARKS:

A DEFtype statement declares that any subsequent untyped variable name beginning with one of the letter(s) specified will be a variable of type <type>. The effect of this statement is to temporarily place the corresponding type declaration character after each untyped variable which has its first letter in <range of letters>. Variables which specify a type declaration character maintain that type. Conflicts are resolved as follows: the type of the untyped variables in the range of letters which conflict will be the type specified in the last executed DEFtype statement that specified that letter.

If no type declaration statements are encountered, PBASIC assumes DEFSNG A-Z.

EXAMPLES:

```
10 DEFDBL L-P
```
All untyped variables beginning with the letters L, M, N, O or P will be double precision variables.

```
10 DEFSTR A
```
All untyped variables beginning with the letter A will be string variables.

```
10 DEFSTR A-Y
20 DEFDBL Z
30 DEFINT I-N
```
All untyped variables beginning with letters from A to H or O to Y will be string variables. Those untyped variables beginning with letters from I to N will be integer variables. All remaining untyped variables (i.e. those starting with a Z) will be double precision variables.

2.8 DEF USR

FORMAT:

DEF USR[<digit>]=<integer expression>

PURPOSE:

To specify the starting address of an assembly language subroutine.

REMARKS:

<digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. For a discussion of the linkage between PBASIC and assembly language routines, see Appendix G.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

EXAMPLE:

```
        .
        .
        .
200 DEF USR0=24000
210 X=USR0(Y^2/2.89)
        .
        .
        .
```

2.9 DELETE

FORMAT:

DELETE[<line number>][-<line number>]

PURPOSE:

To delete program lines.

REMARKS:

PBASIC always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

NOTE:

This command is not allowed within the source program when executing PPROM (see Section 2.37).

EXAMPLES:

DELETE 40	Deletes line 40
DELETE 40-100	Deletes lines 40 through 100, inclusive
DELETE-40	Deletes all lines up to and including line 40

2.10 DIM

FORMAT:

DIM <list of subscripted variables>

PURPOSE:

To specify the maximum values for array variable subscripts and allocate storage accordingly.

REMARKS:

If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 2.34).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

EXAMPLE:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
 .
 .
 .
```

2.11 EDIT

FORMAT:

EDIT <line number>

PURPOSE:

To enter Edit Mode at the specified line.

REMARKS:

In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering the EDIT command for an indirect mode statement, PBASIC types the number of the line to be edited, then it types a space and waits for an EDIT mode subcommand. For a direct mode statement, PBASIC types an exclamation point (!), then a space and waits for an Edit subcommand.

Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, display, delete, replace, or search for text within a line. The subcommands are not echoed. Most of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit Mode subcommands are categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text
6. Ending and restarting Edit Mode

NOTE:

This command is not allowed within the source program when executing PPROM (see Section 2.37).

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, and [i] represents an optional integer (the default is 1).

1. Moving the Cursor

Space — Use the space bar to move the cursor to the right. [i]Space moves the cursor i spaces to the right. Characters are printed as you space over them.

Rubout — In Edit Mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed in reverse order as you backspace over them.

2. Inserting Text

I — I<text> inserts <text> at current cursor position. The inserted characters are printed on the terminal. To terminate insertion, type Escape. If Carriage Return is typed during an Insert command, the effect is the same as typing Escape and then Carriage Return. During an Insert command, the Rubout or Delete key on the terminal may be used to delete characters to the left of the cursor. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Control-G) is typed and the character is not printed.

X — The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, type Escape or Carriage Return.

3. Deleting Text

D — [i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, the remainder of the line is deleted.

H — H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of a line.

4. Finding Text

S — The subcommand [i]S<ch> searches for the ith occurrence of <ch> and positions the cursor before it. The character at the current cursor

position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K — The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. Replacing Text

C — The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you will return to Edit Mode.

6. Ending and Restarting Edit Mode

<cr> — Typing Carriage Return prints the remainder of the line, saves the changes you made, and exits the Edit Mode.

E — The E subcommand has the same effect as Carriage Return, except the remainder of the line is not printed.

Q — The Q subcommand returns to PBASIC command level, without saving any of the changes that were made to the line during Edit Mode.

L — The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, remaining in the Edit Mode. L is usually used to list the line when you first enter Edit Mode.

A — The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

<ESCAPE> — The Escape, or Altmode, key cancels a prefix command, thus aborting a multicharacter command.

NOTE:

If PBASIC receives an unrecognizable command or illegal character while in Edit Mode, it prints a bell (Control-G) and the command or character is ignored.

Syntax Errors

When a Syntax Error is encountered during execution of a program, PBASIC automatically enters Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
Syntax error in 10
10
```

When you finish editing the line and type <CR> or the E subcommand, PBASIC reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination, exit the Edit Mode with the Q subcommand. PBASIC will return to command level, and all variable values will be preserved.

Control-A

To enter Edit Mode on the last line you typed or listed or are currently typing, type Control-A. PBASIC responds with a carriage return, an exclamation point (!) and a space. The cursor will be positioned at the first character in the line. Proceed by typing an Edit Mode subcommand.

Control-A can be used to "move" entire lines by:

1.) LIST line to be moved.
2.) Type Control-A.
3.) Type "I" to enter insert mode.
4.) Type new line number and any corrections.
5.) Type carriage return.

NOTE:

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT ." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line.)

2.12 END

FORMAT:

END

PURPOSE:

To terminate program execution and return to command level.

REMARKS:

END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. PBASIC always returns to command level after an END is executed.

NOTE:

CONT may be used to re-start execution after an END has been reached, provided the program has not been altered.

EXAMPLE:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

2.13 ERASE

FORMAT:

ERASE <variable> [,<variable>]...

PURPOSE:

To deallocate memory space previously allocated for one or more arrays.

REMARKS:

Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Duplicate Definition" error occurs.

EXAMPLE:

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

2.14 ERR AND ERL VARIABLES

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ... Otherwise, use

```
IF ERR = error code THEN ...

IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. PBASIC error codes are listed in Appendix F.

2.15 ERROR

FORMAT:

ERROR <integer expression>

PURPOSE:

1) To simulate the occurrence of a PBASIC error

OR

2) To allow error codes to be defined by the user.
defined by the user.

REMARKS:

The value of <integer expression> must be greater than or equal to 1 and less than or equal to 255. If the value of <integer expression> equals an error code already in use by PBASIC (see Appendix F) the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by PBASIC's error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to PBASIC.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, PBASIC responds with the message Unprintable error. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

EXAMPLE 1:

```
10 S = 10
20 T = 5
30 ERROR S + T
40 END
RUN
String too long in line 30

Or, in direct mode:

Ok
ERROR 15            (you type this line)
String too long     (PBASIC types this line)
Ok
```

EXAMPLE 2:

```
 .
 .
 .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
 .
 .
 .
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
 .
 .
 .
```

2.16 FOR...NEXT

FORMAT:

```
For <variable>=x TO y [STEP z]
  .
  .
  .
NEXT [<variable>]
     or
NEXT <variable>[,<variable>]...
```

where x, y and z are numeric expressions.

PURPOSE:

To allow a series of instructions to be performed in a loop a given number of times.

REMARKS:

<variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. The counter is then incremented by the amount specified by STEP. If the increment is positive, a check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, PBASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value. If the increment is zero, the loop executes indefinitely with <variable> set to be initial value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT

statement may be used for all of them.

The variable in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

EXAMPLE 1:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
1  20
3  30
5  40
7  50
9  60
Ok
```

EXAMPLE 2:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

EXAMPLE 3:

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
OK
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

2.17 GETCLK

FORMAT:

GETCLK [<seconds>][,[<minutes>][,[<hours>][,[<days>][,[<months>]]]]]

Each item is optional and trailing commas need not be specified.

PURPOSE:

To read the current time units from the MDX-BCLK clock hardware.

REMARKS:

All variables in the list are optional. For those that are specified, the numeric value of the corresponding time component is placed into them. Leading and embedded commas are significant since they serve as placeholders.

EXAMPLE:

```
10 GETCLK  A,B,C
20 GETCLK  ,,,DATE (1), DATE (2)
30 PRINT "TIME=";C;":";B;":";A
40 PRINT "DAY="; DATE (1);", MONTH="; DATE (2)
RUN
TIME= 18 : 33 : 57
DAY= 30 , MONTH= 11
```

2.18 GOSUB...RETURN

FORMAT:

```
GOSUB <line number>
  .
  .
  .
RETURN
```

PURPOSE:

To branch to and return from a subroutine.

REMARKS:

<line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause PBASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

EXAMPLE:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

2.19 GOTO

FORMAT:

GOTO <line number>

PURPOSE:

To branch unconditionally out of the normal program sequence to a specified line number.

REMARKS:

If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

EXAMPLE:

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
OK
RUN
R = 5           AREA = 78.5
R = 7           AREA = 153.86
R = 12          AREA = 452.16
Out of DATA in 10
Ok
```

2.20 IF...THEN[...ELSE]
2.21 IF...GOTO

FORMAT:

IF <expression> [,]THEN <statement(s)> / <line number>
ELSE [<statement(s)> / <line number>]

FORMAT:

IF <expression> GOTO <line number>
ELSE [<statement(s)> / <line number>]

PURPOSE:

To make a decision regarding program flow based on the result returned by an expression.

REMARKS:

If the result of <expression> is not zero (logically true), the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero (logically false), the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X THEN PRINT "LESS THAN" ELSE
PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C"
ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line number" error results unless a statement with the specified line number had previously been entered in the indirect mode.

NOTE:

When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

EXAMPLE 1:

```
100 IF(I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
 .
 .
 .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

EXAMPLE 2:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

2.22 INPUT

FORMAT:

INPUT[;][<"prompt string">;]<list of variables>

PURPOSE:

To allow input from the terminal during program execution.

REMARKS:

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to terminate data input does not echo a carriage return/line feed sequence at the terminal.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied by the user must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the messsage "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

If there is only one variable in the INPUT list then responding with only a carriage return causes the variable to be assigned the value of zero if its type is numeric, or a null string if its type is string.

EXAMPLE 1:

```
10 INPUT X
20 PRINT X;"SQUARED IS";X^2
30 END
RUN
? 5              (The 5 was typed in by the user
                 in response to the question mark.)
 5 SQUARED IS 25
Ok
```

EXAMPLE 2:

```
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4 (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

2.23 LET

FORMAT:

[LET] <variable>=<expression>

PURPOSE:

To assign the value of an expression to a variable.

REMARKS:

Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

EXAMPLE:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
.
.
```

or

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
.
.
.
```

2.24 LINE INPUT

FORMAT:

LINE INPUT[;][<"prompt string">;]<string variable>

PURPOSE:

To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

REMARKS:

The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <string variable>.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be aborted by typing Control-C. PBASIC will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

EXAMPLE:

```
20 LINE INPUT  "CUSTOMER INFORMATION? " ; C$
30 PRINT C$
40 GOTO 20

RUN
CUSTOMER INFORMATION? LINDA JONES   234,4 MEMPHIS
LINDA JONES   234,4 MEMPHIS
```

2.25 LIST

FORMAT 1:

LIST [<line number>]

FORMAT 2:

LIST [<line number>[-[<line number>]]]

PURPOSE:

To list to the terminal all or part of the program currently in memory.

REMARKS:

PBASIC always returns to command level after a LIST is executed.

FORMAT 1:

If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program, by typing Control-C, or Control-O, and interrupted by Control-S.) If line number is included then only the specified line will be listed.

FORMAT 2:

This format allows the following options:

1. If only the first number is specified, that line and all higher-numbered lines are listed.

2. If only the second number is specified, all lines from the beginning of the program through that line are listed.

3. If both numbers are specified, the entire range is listed.

NOTE:

See the WIDTH command, Section 2.54, for information on adjusting printed output width.

This command is not allowed within the source program when executing PPROM (see Section 2.37).

EXAMPLES:

Format 1:

LIST	Lists the entire program currently in memory.
LIST 500	Lists line 500.

Format 2:

LIST 150-	List all lines from 150 to the end.
LIST -1000	Lists all lines from the lowest number through 1000.
LIST 150-1000	Lists lines 150 through 1000, inclusive.

2.26 LLIST

FORMAT:

LLIST [<line number>[-[<line number>]]]

PURPOSE:

To list to the line printer all or part of the program currently in memory.

REMARKS:

PBASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST.

NOTE:

See the notes for LIST.

EXAMPLE:

See the examples for LIST.

2.27 LPRINT AND LPRINT USING

FORMAT:

LPRINT [<list of expressions>]
LPRINT USING <"format string">;<list of expressions>

PURPOSE:

To print data at the line printer.

REMARKS:

Same as PRINT and PRINT USING, except output goes to the line printer. See Section 2.38 and Section 2.39.

See the WIDTH command, Section 2.54 for information on adjusting printed output width.

2.28 MID$

FORMAT:

MID$(<string expl>,n[,m])=<string exp2>

where n and m are integer expressions and <string expl> and <string exp2> are string expressions.

PURPOSE:

To replace a portion of one string with another string.

REMARKS:

The characters in <string expl>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string expl>.

EXAMPLE:

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ may also be used as a function that returns a substring of a given string. See Section 3.24.

2.29 NEW

FORMAT:

NEW

PURPOSE:

To delete the program currently in memory and clear all variables.

REMARKS:

NEW is entered at command level to clear memory before entering a new program. PBASIC always returns to command level after a NEW is executed.

NOTE:

This command is not allowed within the source program when executing PPROM (see Section 2.37).

If an application program exists in PROM, when this command is executed an unsuccessful attempt to delete the program will be made and then the program will RUN.

2.30 NULL

FORMAT:

NULL <integer expression>

PURPOSE:

To set the number of nulls to be printed at the end of each line.

REMARKS:

For 10-character-per-second tape punches, <integer expression> should be >=3. When tapes are not being punched, <integer expression> should be 0 or 1 for Teletypes and Teletype-compatible CRTs. <integer expression> should be 2 or 3 for 30 cps hard copy printers. The default value is 0; the maximum is 255.

EXAMPLE:

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X<50 GOTO 800
.
.
.
```

Two null characters will be printed after each line.

2.31 ON ERROR GOTO

FORMAT:

ON ERROR GOTO <line number>

PURPOSE:

To enable error trapping and specify the first line of the error handling subroutine.

REMARKS:

Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes PBASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE:

If an error occurs during execution of an error handling subroutine, the PBASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

See the RESUME instruction, Section 2.45 for information on how to return control to the user program from an error trapping routine.

EXAMPLE:

```
10 ON ERROR GOTO 1000
```

2.32 ON...GOSUB
2.33 ON...GOTO

FORMAT:

ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

PURPOSE:

To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

REMARKS:

The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

EXAMPLE:

```
100 ON L-1 GOTO 150,300,320,390
```

2.34 OPTION BASE

FORMAT:

OPTION BASE n
where n is 1 or 0

PURPOSE:

To declare the minimum value for array subscripts.

REMARKS:

The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

2.35 OUT

FORMAT:

OUT I,J
where I and J are integer expressions in the range 0 to 255.

PURPOSE:

To send a byte to a machine output port.

REMARKS:

The integer expression I is the port number, and the integer expression J is the data to be transmitted. I and J must evaluate to integers <=255 (HEX FF).

NOTE:

Port addresses E8-EB HEX are "Illegal function call"s when using PBASIC/9511 version, because they are dedicated to the 9511 board. These ports should not be used in the non-9511 version to assure compatability between the two versions.

EXAMPLE:

```
100 OUT 32,100
```

2.36 POKE

FORMAT:

POKE I,J
where I and J are integer expressions

PURPOSE:

To write a byte into a memory location.

REMARKS:

The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255. I must be in the range of 0 to 65535.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Section 3.26.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

NOTE:

POKE can cause fatal damage either to PBASIC, user program, or the other RAM Areas in memory if indiscriminately used.

EXAMPLE:

```
10 POKE &H5A00,&HFF
```

2.37 PPROM

FORMAT:

PPROM [<type>]

PURPOSE:

This command will initiate the programming of a user's PBASIC application program into EPROMs placed in the PPG hardware device. The entire PBASIC program will be segmented and burned into the required number of EPROMs. The user will be prompted when it is time to insert and remove each EPROM. The only optional argument is the type of PROM (2708, 2716, and 2758) being programmed with a default of 2716. The user is able to abort the operation at any time via Control-C.

REMARKS:

<type>

The user may choose the type of EPROM chip to use with the type option. Choices are 2708, 2716, or 2758. The default is 2716. To specify a type the user must enter the last two digits of the type number.

```
To use 2708s enter:     PPROM 08
To use 2758s enter:     PPROM 58
To use 2716s enter:     PPROM 16 or PPROM
```

Any deviation from this will produce a Syntax error.

TO USE:

1) Enter PPROM program command, optional type and (CR).

2) The user will be prompted with an astrick sign (*) in the next line, left most cursor position. This is followed by the sequence number of the chip and the number of chips required for the entire program.

3) Insert blank EPROM chip and enter (CR).

4) A bell (Control-G) is sounded at the completion of programming each chip.

5) A prompt will appear again if more chips are needed.

6) Control-C will abort this command at any point.

EXAMPLE:

If the user has a 7K program and wishes to burn the application into 2716s, the following would be done:

```
PPROM           (user enters to program 2716)
*1 / 4          4 chips are needed, user performs step 3
*2 / 4          user performs step 3
*3 / 4          user performs step 3
*4 / 4          user performs step 3
Ok              finished
```

ERROR CONDITIONS:

1) When the user places an EPROM in the PPG device, the chip must be blank. If it is not, an error condition will occur:

'***** ERROR - CHIP NOT BLANK *****

2) After each program segment is burned into the chip, the data is verified. If the verification reveals bad data an error condition will occur:

'****** ERROR - CHIP DATA DOES NOT VERIFY ******

USER OPTIONS:

A) Replace chip with an erased chip and enter (CR). This will cause PBASIC to continue the command, not start over.

(OR)

B) Enter control C. This will abort the command.

(OR)

C) Anything else will cause a syntax error. (However, the command will not abort. The user will be reprompted for the same chip number.)

3) The error:

*****ERROR-DEBUG COMMAND WITHIN SOURCE AT LINE NUMBER XX ******

will occur when any of the following commands reside in source when PPROM is executed.

AUTO	NEW
DELETE	RENUM
EDIT	RUN
LIST	PPROM
LLIST	RPROM

PBASIC then returns to the direct mode for source alteration.

4) Other error conditions may occur but can not be

detected by the interpreter, such as:

A) The user inserts a type chip other than specified. [ie. 2708 specified and 2716 is inserted.]

(OR)

B) The user inserts the chip backward.

NOTE:

The PPROM command is not allowed within the source program when executing I

CAUTION:

In order to prevent possible destruction of PROMs, the user should not insert PROMs in the ZIP DIP socket until prompted. Also, the user should insure that PROM TYPE SWITCH is in the correct position for type of PROM being used. The switch should be in the 2708 position for 2708 PROMs and in the 2716 position for 2716 and 2758 PROMs. If a PROM is inserted in the ZIP DIP socket with the PROM TYPE SWITCH in the wrong position, the PROM may be subjected to voltages which could destroy it.

2.38 PRINT

FORMAT:

PRINT [<list of expressions>]

PURPOSE:

To output data at the terminal.

REMARKS:

If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. PBASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly.

If the list of expressions terminates without a comma or a semicolon, a carriage return, line feed is generated at the end of the line. If the printed line is longer than the terminal width, PBASIC inserts a carriage return and line feed and continues printing.

This rule applies no matter where the cursor is positioned by direct cursor addressing. Therefore, ensure that a PRINT command is placed in the logic to periodically reset this width counter to prevent PBASIC from inserting extra CR, LFs into screen text or use WIDTH 255, see Section 2.54.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 10^(-7) is output as .0000001 and 10^(-8) is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-15 is output as .000000000000001 and 1D-16 is output as 1D-16.

EXAMPLE 1:

```
10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
 10            0            -25            3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

EXAMPLE 2:

```
LIST
10 INPUT X
20 PRINT X;"SQUARED IS";X^2;"AND";
30 PRINT X;"CUBED IS";X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

A question mark may be used in place of the word PRINT in a PRINT statement.

EXAMPLE 3:

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

2.39 PRINT USING

FORMAT:

PRINT USING <"format string">;<list of expressions>

PURPOSE:

To print strings or numbers using a specified format.

REMARKS and EXAMPLES:

<list of expressions> is comprised of the string and expressions or numeric expressions that are to be printed, separated by semicolons. <"format string"> is a string expression comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!"
Specifies that only the first character in the given string is to be printed.

"\n spaces\"
Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

```
10 A$="LOOK":B$="OUT":C$="\   \"
20 PRINT USING "!";A$;B$
30 PRINT USING "\ \";A$;B$
40 PRINT USING C$;A$;B$;"!!"
Ok
RUN
LO
LOOOUT
LOOK OUT  !!
```

"&"
Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input.

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

"#"
A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

"."
A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.## ";10.2,5.3,66.789,.234
10.20  5.30 66.79  0.23
```

In the last example, one space was inserted at the end of the format string to separate the printed values on the line.

"+"
A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

"-"
A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.## ";-68.95,2.4,55.6,-.9
-68.95  +2.40 +55.60  -0.90

PRINT USING "##.##- ";-68.95,22.449,-7.01
68.95- 22.45   7.01-
```

"**"
A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.# ";12.39,-0.9,765.1
*12.4 *-0.9 765.1
```

"$$"
A double dollar sign causes a dollar sign to be printed to the

immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
 $456.78
```

"**$"

The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

","

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.

```
PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "####.##,";1234.5
1234.50,
```

"^^^^"

Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
 2.35E+02

PRINT USING ".####^^^^-";888888
 .8889E+06

PRINT USING "+.##^^^^";123
+.12E+03
```

"_"

An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "__" in the format string.

NOTE:

If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

2.40 RANDOMIZE

FORMAT:

RANDOMIZE [<expression>]

PURPOSE:

To reseed the random number generator.

REMARKS:

If <expression> is omitted, PBASIC suspends program execution and asks for a value by printing

Random Number Seed (0-65535)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

EXAMPLE:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
Ok
RUN
Random Number Seed (0-65535)? 3 (user types 3)
 .88598  .484668  .586328  .119426  .709225
Ok
RUN
Random Number Seed (0-65535)? 4 (user types 4 for new sequence)
 .803506  .162462  .929364  .292443  .322921
Ok
RUN
Random Number Seed (0-65535)? 3 (same sequence as first RUN)
 .88598  .484668  .586328  .119426  .709225
Ok
```

2.41 READ

FORMAT:

READ <list of variables>

PURPOSE:

To read values from a DATA statement and assign them to variables. (See DATA, Section 2.5)

REMARKS:

A READ statement must always be used in conjunction with a DATA statement. READ statements assign DATA statement values to variables on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result, pointing to the DATA statement which did not correspond with the format specified in the READ.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statment. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an Out of DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start or beginning at any designated DATA statement, use the RESTORE statement (see RESTORE, Section 2.44)

EXAMPLE 1:

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

EXAMPLE 2:

```
LIST
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY            STATE           ZIP
DENVER,         COLORADO        80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

2.42 REM

FORMAT:

```
REM <remark>
    or
' <remark>
```

PURPOSE:

To allow explanatory remarks to be inserted in a program.

REMARKS:

REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

In PBASIC, remarks may be added to the end of a line by preceding the remark with a single quotation mark (') instead of :REM.

EXAMPLE:

```
      .
      .
      .
      .
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20          :REM BEGIN LOOP
140 SUM=SUM + V(I)
      .
      .
      .
or
      .
      .
      .
120 FOR I=1 TO 20          'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)           'BEGIN LOOP
140 NEXT I
      .
      .
      .
```

2.43 RENUM

FORMAT:

RENUM [[<new number>][,[<old number>][,<increment>]]]

PURPOSE:

To renumber program lines.

REMARKS:

<new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

NOTE:

This command is not allowed within the source program when executing PPROM (see Section 2.37).

RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

EXAMPLES:

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will increment by 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

2.44 RESTORE

FORMAT:

RESTORE [<line number>]

PURPOSE:

To allow DATA statements to be reread from a specified point.

REMARKS:

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

EXAMPLE:

```
10 READ A
20 READ B
30 RESTORE
40 READ C
50 DATA 57,68,79
60 PRINT A,B,C
70 END
RUN
 57            68            57
Ok
```

2.45 RESUME

FORMATS:

```
RESUME
RESUME 0
RESUME NEXT
RESUME <line number>
```

PURPOSE:

To continue program execution after an error recovery procedure has been performed via 'ON ERROR'.

REMARKS:

Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME <line number>	Execution resumes at <line number>.

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

EXAMPLE:

```
10 ON ERROR GOTO 900

   .
   .
   .
900 IF (ERR=230)AND(ERL=90) THEN PRINT "TRY
AGAIN":RESUME 80
   .
   .
   .
```

2.46 RPROM

FORMAT:

RPROM [<type>]

PURPOSE:

This command will initiate the reading of a user's PROMed application, placed in the PPG hardware device, into RAM memory. The entire segmented PROMed program is read into memory in sequence and terminated after the last EPROM has been read. The user will be prompted when to extract and insert each EPROM. The only optional argument is the type of EPROM (2708, 2716, and 2758) being programmed with a default of 2716. The user is able to abort the operation at any time via a Control-C.

REMARKS:

<type>

The user may choose the type of EPROM chip to use with the type option. Choices are 2708, 2716, or 2758. The default is 2716. To specify a type the user must enter the last two digits of the type number.

To use 2708s enter:	RPROM 08
To use 2758s enter:	RPROM 58
To use 2716s enter:	RPROM 16, or RPROM

TO USE:

1) Enter RPROM command, optional type (default is 2716), and (CR).

2) The user will be prompted for the first chip to be inserted with a number sign (#) followed by the number one (1).

3) The chip is read into RAM. Since the first chip contains a checksum table as well as the total number of chips in the program it is essential it be entered first. From this point on the user will be prompted for the next chip number and the total number of chips.

4) Control-C will abort the command at any point.

5) If the command is aborted before all the chips have been read, the data previously read will not remain in memory.

6) A maximum limit of XX PROMs per program will

be allowed. When this limit is exceeded the PBASIC error message 'Out of memory' will appear on the terminal and the command will abort. The source previously read (PROMs 1 thru (XX-1)) will not remain in RAM.

XX = FRE(0) / PROM chip size

EXAMPLE:

If the user has a program on 3, 2716 EPROMs and wishes to read the program into RAM, the following would be done:

```
RPROM          ; user enters command (defaults to 2716 type)
# 1            ; system prompts
               ; user inserts first chip and enters (CR)
# 2 / 3        ; when the chip has been read the system prompts
               ; user inserts second chip and enters (CR)
# 3 / 3        ; when the chip has been read the system prompts
               ; user inserts third and final chip and enters (CR)
Ok             ; command is finished
```

ERROR CONDITIONS:

1) Reading too many PROMs causes the ERROR condition:

'Out of memory'

Source read from previously read PROMs will not remain in RAM. The RPROM command will be aborted.

USER OPTIONS:

A) Re-execute the command.

2) Chips are inserted out of sequence. This will result in a

'OUT OF SEQUENCE --- Chip # X.

USER OPTIONS:

A) Insert chip X. No previously read data is lost.

3) Other error conditions that might occur, that are not detectable by PBASIC.

A) The user inserts a type chip other than specified. (i.e. 2708 specified and 2716 is inserted)

NOTE:

This command is not allowed within the source program when executing PPROM (see Section 2.37).

CAUTION:

In order to prevent possible destruction of PROMs, the user should not insert PROMs in the ZIP DIP socket until prompted. Also, the user should insure that PROM TYPE SWITCH is in the correct position for type of PROM being used. The switch should be in the 2708 position for 2708 PROMs and in the 2716 position for 2716 and 2758 PROMs. If a PROM is inserted in the ZIP DIP socket with the PROM TYPE SWITCH in the wrong position, the PROM may be subjected to voltages which could destroy it.

2.47 RUN

FORMAT:

RUN [<line number>]

PURPOSE:

To execute the program currently in memory.

REMARKS:

If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. PBASIC always returns to command level after the program has finished executing.

NOTE:

RUN may be specified as either a direct or indirect statement in a RAM system but this command is not allowed within the source program when executing PPROM (see Section 2.37).

EXAMPLE:

```
RUN 20  'STARTS EXECUTION AT STATEMENT 20
```

2.48 SETCLK

FORMAT:

SETCLK [<seconds>][,[<minutes>][,[<hours>][,[<days>][,[<months>]]]]]

Each item is optional and trailing commas need not be specified.

PURPOSE:

To set specified time components of the MDX-BCLK clock hardware to specific values.

REMARKS:

Each item in the list must evaluate to an integer expression. If an item is omitted, then its corresponding time component remains unchanged. Each item is checked for the proper range as well as consistency with other items. The ranges are:

seconds:	0-59
minutes:	0-59
hours:	0-23
days:	1-31
months:	1-12

The day of the month is compared with the maximum number of days in that month and will result in an Error #5 if exceeded.

EXAMPLE:

```
10  PRINT  "CURRENT DATE IS ";DATE$
20  LINE INPUT "CHANGE DATE (Y/N)?";ANSWER$
30  IF ANSWER$ <> "Y" THEN 70
40      INPUT "NEW DAY NUMBER";DAY
50      INPUT "NEW MONTH NUMBER";MONTH
60      SETCLK   ,,,DAY,MONTH
70  REM ENDIF
80  PRINT "CURRENT TIME IS ";TIME$
90  LINE INPUT "CHANGE TIME (Y/N)?";ANSWER$
100 IF ANSWER$ <> "Y" THEN 160
110     INPUT "NEW HOUR IS";HOUR
120     INPUT "NEW MINUTE IS";MINUTE
130     INPUT "NEW SECOND IS";SECOND
140     LINE INPUT "HIT <CR> WHEN NEW TIME IS REACHED";REPLY$
150     SETCLK   SECOND,MINUTE,HOUR
160 REM ENDIF
```

2.49 STOP

FORMAT:

STOP

PURPOSE:

To terminate program execution and return to the command level.

REMARKS:

STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in nnn where nnn is STOP statement line number.

PBASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 2.4).

EXAMPLE:

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
Break in 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

2.50 SWAP

FORMAT:

SWAP <variable>,<variable>

PURPOSE:

To exchange the values of two variables.

REMARKS:

Any type variable may be SWAPped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

EXAMPLE:

```
LIST

10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
ONE FOR ALL
ALL FOR ONE
Ok
```

2.51 TRON/TROFF

FORMAT:

```
TRON
TROFF
```

PURPOSE:

To trace the execution of program statements.

REMARKS:

As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

EXAMPLE:

```
TRON

Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1  10  20
[50][60][30][40] 2  20  30
[50][60][70]
Ok
TROFF
Ok
```

2.52 WAIT

FORMAT:

WAIT <port number>, I[,J]
where I and J are integer expressions

PURPOSE:

To suspend program execution while monitoring the status of a machine input port.

REMARKS:

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, PBASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.

CAUTION:

It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine. Control-C will not abort this command while in a port wait loop.

Port addresses &HE8-&HEB are "Illegal function call"s when using the PBASIC/9511 version, because they are dedicated for the MDX-Math(9511) board operations.

EXAMPLE:

```
100 WAIT 32,2
```

2.53 WHILE...WEND

FORMAT:

```
WHILE <expression>
    .
    .
[<loop statements>]
    .
    .
WEND
```

PURPOSE:

To execute a series of statements in a loop as long as a given condition is true.

REMARKS:

If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. PBASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

EXAMPLE:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115       FLIPS=0
120       FOR I=1 TO J-1          :REM J=number of A$ elements
130             IF A$(I)>A$(I+1) THEN
                        SWAP A$(I),A$(I+1):FLIPS=1
140       NEXT I
150 WEND
```

2.54 WIDTH

FORMAT:

WIDTH [LPRINT] <integer expression>

PURPOSE:

To set the printed line width in number of characters for the terminal or line printer.

REMARKS:

If the LPRINT option is omitted, the line width of the terminal is set. If LPRINT is included, the line width of the line printer is set. The default width for the terminal is 80 characters, and the default width for the line printer is 132 characters.

<integer expression> must have a value in the range 0 to 255. If <integer expression> is 255, the line width is "infinite," that is, PBASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255. If zero (0) width is used, a blank line will be generated before the line is output.

EXAMPLE:

```
10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

2.55 WRITE

FORMAT:

WRITE[<list of expressions>]

PURPOSE:

To output data at the terminal.

REMARKS:

If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output to the terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, PBASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement, Section 2.38.

EXAMPLE:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
80,90,"THAT'S ALL"
Ok
```

CHAPTER 3

PBASIC FUNCTIONS

The intrinsic functions provided by PBASIC are presented in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y Represent any numeric expressions

I and J Represent integer expressions

X$ and Y$ Represent string expressions

If a floating point value is supplied where an integer is required, PBASIC will round the fractional portion and use the resulting integer.

3.1 ABS

FORMAT:

ABS(X)

ACTION:

Returns the absolute value of the expression X.

EXAMPLE:

```
PRINT ABS(7*(-5))
 35
Ok
```

3.2 ASC

FORMAT:

ASC(X$)

ACTION:

Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix H for ASCII codes.) If X$ is null, an "Illegal function call" error is returned.

EXAMPLE:

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
```

See the CHR$ function for ASCII-to-string conversion.

3.3 ATN

FORMAT:

ATN(X)

ACTION:

Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

EXAMPLE:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.24905
Ok
```

3.4 BCD

FORMAT:

BCD(X)

ACTION:

This function will convert a 16-bit binary number to its BCD equivalent representation. The argument may be a constant or the result of an arithmetic expression.

EXAMPLE:

```
PRINT BCD(10)
 16
```

The value 10 (0000 0000 0000 1010) is converted to the value 10 in BCD format (0000 0000 0001 0000, or &H0010).

3.5 BIN

FORMAT:

BIN(X)

ACTION:

This function converts a number represented in BCD to its 16-bit binary equivalence. The argument may be a constant or the result of an arithmetic expression. The maximum BCD value is &H9999.

EXAMPLE:

```
PRINT BIN(16)
 10
```

The BCD number 16 (0000 0000 0001 0000 or &H0010) is converted to the binary equivalent 10 (0000 0000 0000 1010).

3.6 CDBL

FORMAT:

CDBL(X)

ACTION:

Converts X to a double precision number.

NOTE:

Conversion from single percision to double percision only allows 7 digits of accuracy to be transferred to the double percision variable.

EXAMPLE:

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67   454.6700134277344
Ok
```

3.7 CHR$

FORMAT:

CHR$(I)

ACTION:

Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix H.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message, or a form feed could be sent (CHR$(12)) to clear a CRT screen and return the cursor to the home position or to force top-of-form on the printer.

EXAMPLE:

```
PRINT CHR$(66)
B
Ok
```

See the ASC function for ASCII-to-numeric conversion.

3.8 CINT

FORMAT:

CINT(X)

ACTION:

Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.

EXAMPLE:

```
PRINT CINT(45.67)
 46
Ok
```

See the CDBL and CSNG functions for converting numbers to the double precision and single precision data type. See also the FIX and INT functions, both of which return integers.

3.9 COS

FORMAT:

COS(X)

ACTION:

Returns the cosine of X expressed in radians. The calculation of COS(X) is performed in single precision.

EXAMPLE:

```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

3.10 CSNG

FORMAT:

CSNG(X)

ACTION:

Converts X to a single precision number.

EXAMPLE:

```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
 975.3421   975.342
Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

3.11 DATE$

FORMAT:

DATE$

ACTION:

Returns the current day and month in string form. These values are obtained from the MDX-BCLK clock hardware.

EXAMPLE:

```
PRINT DATE$
30-NOV
```

3.12 EXP

FORMAT:

EXP(X)

ACTION:

Returns e to the power of X. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

EXAMPLE:

```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
Ok
```

3.13 FIX

FORMAT:

FIX(X)

ACTION:

Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

EXAMPLES:

```
PRINT FIX (58.75)
 58
Ok
PRINT FIX(-58.75)
-58
Ok
```

3.14 FRE

FORMAT:

FRE(0)

FRE(X$)

ACTION:

Arguments to FRE are dummy arguments. If the argument is 0 (numeric), FRE returns to the terminal the number of bytes in memory not being used by PBASIC. If the argument is a string, FRE returns the number of free bytes in the string.

EXAMPLE:

```
10 PRINT   FRE(0)
20 DIM     A$(1000)
30 PRINT   FRE(0)
40 CLEAR
50 PRINT   FRE(0)
60 END
RUN
 24473
 21461
 24473
Ok
```

3.15 HEX$

FORMAT:

HEX$(X)

ACTION:

Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX$(X) is evaluated. X must be < = 65535.

EXAMPLE:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X;"DECIMAL IS ";A$;" HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT$ function for octal conversion.

3.16 INP

FORMAT:

INP(I)

ACTION:

Returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement, Section 2.35.

NOTE:

Port addresses E8-EB HEX are "Illegal function call"s when using PBASIC/9511 version, because they are dedicated to the MDX-Math board operations.

EXAMPLE:

```
100 A=INP(255)  'Set A to value of Port 255
```

3.17 INPUT$

FORMAT:

INPUT$(X)

ACTION:

Returns a string of X characters, read from the terminal. No characters will be echoed and all control characters are passed through except Control-C, which is used to interrupt the execution of the INPUT$ function.

EXAMPLE:

```
.
.
.
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
.
.
.
```

3.18 INSTR

FORMAT:

INSTR([I,]X$,Y$)

ACTION:

Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 1 to 255. If I>LEN(X$) or if X$ is null or if Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions or string literals.

EXAMPLE:

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

3.19 INT

FORMAT:

INT(X)

ACTION:

Returns the largest integer <=X.

EXAMPLES:

```
PRINT INT(99.89)
 99
Ok
PRINT INT(-12.11)
-13
Ok
```

See the FIX and CINT functions which also return integer values.

3.20 LEFT$

FORMAT:

LEFT$(X$,I)

ACTION:

Returns a string comprised of the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

EXAMPLE:

```
10 A$ = "PROCESS BASIC"
20 B$ = LEFT$(A$,7)
30 PRINT B$
PROCESS
Ok
```

Also see the MID$ and RIGHT$ functions.

3.21 LEN

FORMAT:

LEN(X$)

ACTION:

Returns the number of characters in X$. Non-printing characters and blanks are counted.

EXAMPLE:

```
10 X$ = "CARROLLTON, TEXAS"
20 PRINT LEN(X$)
RUN
 17
Ok
```

3.22 LOG

FORMAT:

LOG(X)

ACTION:

Returns the natural logarithm of X. X must be greater than zero.

EXAMPLE:

```
PRINT LOG(45/7)
 1.86075
Ok
```

3.23 LPOS

FORMAT:

LPOS(X)

ACTION:

Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

EXAMPLE:

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

3.24 MID$

FORMAT:

MID$(X$,I[,J])

ACTION:

Returns a string of length J characters from X$ beginning with the Ith character. I must be in the range 1 to 255 and J must be in the range 0 to 255. If J is ommitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), MID$ returns a null string.

EXAMPLE:

```
LIST
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
RUN
GOOD EVENING
Ok
```

Also see the LEFT$ and RIGHT$ functions.

3.25 OCT$

FORMAT:

OCT$(X)

ACTION:

Returns a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated. X must be in the range 0 to 65535.

EXAMPLE:

```
PRINT OCT$(24)
30
Ok
```

See the HEX$ function for hexadecimal conversion.

3.26 PEEK

FORMAT:

PEEK(I)

ACTION:

Returns the byte (decimal integer in the range 0 to 255) read from memory location I. I must be in the range 0 to 65535. PEEK is the complementary function to the POKE statement, Section 2.36.

EXAMPLE:

```
PRINT PEEK (&H6023)
 0
Ok
```

3.27 POS

FORMAT:

POS(I)

ACTION:

Returns the column number of the cursor position in the current print line. The leftmost position is 1. I is a dummy argument.

EXAMPLE:

```
IF POS(I)>60 THEN PRINT CHR$(13)
```

Also see the LPOS function.

3.28 RIGHT$

FORMAT:

RIGHT$(X$,I)

ACTION:

Returns the rightmost I characters of string X$. If I>=LEN(X$), returns X$. If I=0, the null string (length zero) is returned.

EXAMPLE:

```
10 A$="PROCESS BASIC"
20 PRINT RIGHT$(A$,5)
RUN
BASIC
Ok
```

Also see the MID$ and LEFT$ functions.

3.29 RND

FORMAT:

RND[(X)]

ACTION:

Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE, Section 2.40). However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

EXAMPLE:

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 24  30  31  51  5
Ok
```

3.30 ROTATE

FORMAT:

ROTATE (X,Y)

ACTION:

This function will rotate a 16-bit argument in a circular manner with the direction and number of binary places specified by a second 8-bit argument. The direction of rotation is determined by the sign of the second argument with a positive argument signifying a right rotate and a negative argument signifying a left rotate. The absolute value of the second argument determines the number of bit positions to rotate. Either argument could be a constant or the result of an arithmetic expression.

EXAMPLE:

```
PRINT ROTATE (1,2)
 16384
```

The value 1 (0000 0000 0000 0001) was rotated 2 places to the right thus producing the result 16384 (0100 0000 0000 0000).

3.31 SGN

FORMAT:

SGN(X)

ACTION:

Returns a value 1, 0, or -1 based on the sign of X.

If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

EXAMPLE:

```
10 ON SGN(X)+2 GOTO 100,200,300
 .
 .
 .
100                                 'X is negative
 .
 .
 .
200                                 'X is 0
 .
 .
 .
300                                 'X is positive
```

3.32 SHIFT

FORMAT:

SHIFT (X,Y)

ACTION:

This function will shift a 16-bit argument horizontally, the direction and number of binary places specified by a second 8-bit argument. The direction of shifts is determined by the sign of the second argument with a positive argument signfying a right shift and a negative argument signifying a left shift. The absolute value of the second argument determines the number of bit positions to shift. All vacated bits on a shift operation are zero (0) filled. Either argument could be a constant or the result of an arithmetic expression.

EXAMPLE:

```
PRINT SHIFT (84,-3)
 672
```

The bits of the value 84 (0000 0000 0101 0100) were shifted left 3 times resulting in the value 672 (0000 0010 1010 0000).

3.33 SIN

FORMAT:

SIN(X)

ACTION:

Returns the sine of X expressed in radians. SIN(X) is calculated in single precision. COS(X)=SIN(X+3.14159/2).

EXAMPLE:

```
PRINT SIN(1.5)
 .997495
Ok
```

3.34 SPACE$

FORMAT:

SPACE$(X)

ACTION:

Returns a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.

EXAMPLE:

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
Ok
```

Also see the SPC function.

3.35 SPC

FORMAT:

SPC(I)

ACTION:

Prints I(MOD 80) blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255.

EXAMPLE:

```
PRINT "OVER";SPC(15);"THERE"
OVER               THERE
Ok
```

Also see the SPACE$ function.

3.36 SQR

FORMAT:

SQR(X)

ACTION:

Returns the square root of X. X must be >=0.

EXAMPLE:

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.16228
 15            3.87298
 20            4.47214
 25            5
Ok
```

3.37 STR$

FORMAT:

STR$(X)

ACTION:

Returns a string representation of the value of X.

EXAMPLE:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
.
.
.
```

Also see the VAL function.

3.38 STRING$

FORMATS:

STRING$(I,J)

STRING$(I,X$)

ACTION:

Returns a string of length I whose characters all have ASCII code J or the first character of X$. 0<=I<255, 0<=J<256. Values of J>=128 are printed as if the Most Significant Bit (MSB) were dropped.

EXAMPLE:

```
10 X$ = STRING$(10,45)
20 PRINT X$;"MONTHLY REPORT";X$
RUN
----------MONTHLY REPORT----------
Ok
```

3.39 TAB

FORMAT:

TAB(I)

ACTION:

Spaces to position I on the terminal. If the current print position is already beyond space I, then printing resumes in Column I on the next line. Space 1 is the leftmost position, and the rightmost position is the width of the device. For PRINT, I is MOD 80; for LPRINT, I is MOD 132. I must be in the range 0 to 255. TAB may only be used in PRINT and LPRINT statements.

EXAMPLE:

```
10 PRINT "NAME";TAB(25);"AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$;TAB(25);B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                      AMOUNT

G. T. JONES               $25.00
Ok
```

3.40 TAN

FORMAT:

TAN(X)

ACTION:

Returns the tangent of X expressed in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

EXAMPLE:

```
10 Y = Q*TAN(X)/2
```

3.41 TIME$

FORMAT:

TIME$

ACTION:

Returns the current time (HH:MM:SS) in string form. These values are obtained from the MDX-BCLK clock hardware.

EXAMPLE:

```
PRINT TIME$
18:33:57
```

3.42 USR

FORMAT:

USR[<digit>](X)

ACTION:

Calls the user's assembly language subroutine with the argument X. <digit> is in the range of 0-9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See Appendix G for a dicussion of the linkage between PBASIC and assembly language routines.

EXAMPLE:

```
10 DEF USR1= &HC000
20 DEF USR2= &HD000
30 A = T*SIN (Y)
40 B = USR1  (A)        'Compute using User Function 1
50 C = USR2  (A)        'Compute using User Function 2
 .
 .
 .
```

3.43 VAL

FORMAT:

VAL(X$)

ACTION:

Returns the numerical value of string X$. If the first character of X$ is not +, -, &, or a digit, VAL(X$)=0.

EXAMPLE:

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN
PRINT NAME$ TAB(25) "LONG BEACH"
.
.
.
```

See the STR$ function for numeric to string conversion.

3.44 VARPTR

FORMAT:

VARPTR (<variable name>)

ACTION:

Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array), and the address returned will be an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR (A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

NOTE:

All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

EXAMPLE:

```
100 X=USR (VARPTR(Y))
```

APPENDIX A

SYSTEM CONFIGURATION

A.1 MEMORY LAYOUT

A.1.1 PBASIC INTERPRETER

The PBASIC Interpreter can execute from either RAM or ROM and resides at 0000 up to 8000 HEX, as shown in Figure A-1. PBASIC requires RAM memory in locations F000 to FFFF HEX for interpreter variables and stack space. This comprises the main configuration of the PBASIC interpreter with execution starting at 0000 HEX.

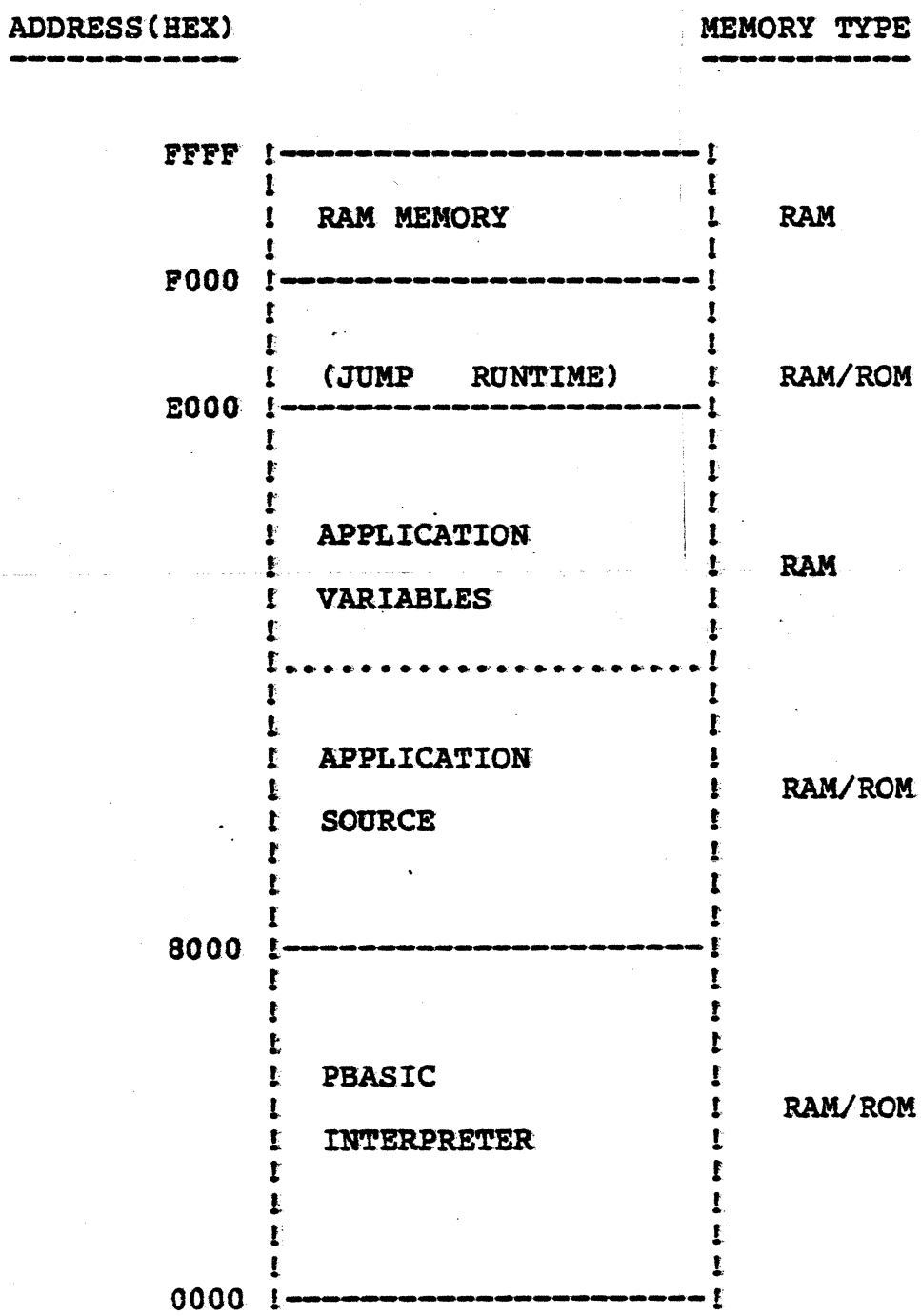
An automatic Runtime system can be configured by placing the AUTO RUN PROM at E000 HEX and starting execution at E000 HEX by power-on or reset. This will automatically start to RUN the application program that currently resides in the source area at 8000 HEX.

A.1.2 PBASIC APPLICATION PROGRAM

The PBASIC application program resides in RAM or PROM starting at location 8000 HEX and can extend upward to location EFFF HEX, or DFFF for automatic Runtime, as shown in Figure A-1. This area is dedicated for both a program's source statements, in RAM or ROM, and program variables, always RAM, giving the user a maximum limit of 28k bytes of application memory space.

FIGURE A-1

PBASIC SYSTEM CONFIGURATION



A.2 HARDWARE CONFIGURATION

The minimum hardware and associated PORT assignments required to operate PBASIC is as follows:

MD BOARD	PBASIC DEDICATED PORT (HEX)
MDX-CPU I OR II	7C-7F (CTC)
MDX-EPROM/UART	DC-DF (UART)
MDX-EPROM/ROM	
MDX-DRAM 8/16/32	
MDX-PIO	D0-D3 (LP)
	D4-D7 (PPG)
MDX-BCLK	70-71 (CLK)

For PBASIC/9511 version only:

MDX-MATH	E8-EB (PIO)

WARNING- user is blocked from using port I/O at the MATH board ports due to contention problems.

A.3 BOARD LEVEL DOCUMENTATION

If MDX boards are used, then the appropriate Operations Manual for that board must be referenced for functional operation.

APPENDIX B

SYSTEMS GENERATION

B.1 PBASIC INTERPRETER

B.1.1 INITIAL PROGRAM DEVELOPMENT

The user starts execution of the interpreter, at location &H0000, in the Development configuration. In this configuration, RAM memory is required from &H8000 up to &HEFFF as needed. RAM memory is initialized upon power-up of PBASIC in this configuration. PBASIC statements are entered via the terminal, utilizing the full capabilities of the PBASIC interpreter. This allows program statements to be placed into the source program area.

Once the program has been input, the user can then RUN the application program that resides in RAM. The application program can be debugged thoroughly and any edits or modification can be made until program testing is completed.

B.1.2 PROMMING APPLICATION PROGRAM

Once the program has been tested, the program can be burned into PROMs via the PPROM command (See Section 2.37). The PROM application is then placed in the same memory space as the RAM version. The only adjustment the user must make in running the PROM version is that RAM variable storage will begin at the next even 1000 HEX boundary from where the application program ended.

EXAMPLE:

If Application Program Memory is &H8000 to &H9700
then variable storage RAM starts at &HA000

B.1.3 RUNNING PROM APPLICATION IN DEVELOPMENT CONFIGURATION

The user can then power-up PBASIC and type in RUN to run the PROMed application program. The application program is now in a non-volatile state and power can be removed from the system without destroying the application program.

B.1.4 READING PROMMED APPLICATION INTO RAM

The user places RAM memory in the application program space &H8000 to &HEFFF as required. The RPROM command (See Section 2.46) is used to read the PROMed application program, in sequence, back into RAM memory. The user is back to the initial program development step with the current application program residing in RAM memory.

B.1.5 AUTOMATIC RUNTIME MODE

To use the Automatic Runtime Mode, the user straps the CPU to start executing code at location &HE000, where the Runtime PROM resides. With the PBASIC interpreter and application program in non-volatile memory, when power is applied, PBASIC will automatically start running the application program that resides in the system.

Note that an application system requiring more than 24k bytes of memory can only be run in the development mode because of memory requirements.

APPENDIX C

INTEGRITY TEST

C.1 INTEGRITY TEST PURPOSE

The purpose of the Integrity Test is to assure continuity and integrity of the PROMed program sequence. The PROM application is checked when the RUN command is invoked and the error message

***** CHECKSUM ERROR IN CHIP #XX *****

is output to the terminal to notify the user that chip # XX is bad, out of order, or missing. No PROM program will execute until the entire application program passes the Integrity Test.

C.2 INTEGRITY TEST SCHEME

Every PBASIC application program contains a Checksum Table which preceeds the program source. This 38 byte table, as shown in Figure C-1, is dedicated to a checksum byte for each PROM image, plus additional information. When the application program is placed into PROM, a checksum byte for each 1K or 2K bytes, according to which type of PROM(s) is used, is placed into the checksum table according to its appropriate chip number. Also contained within the table is the address of where variable RAM storage is expected to start and the number and type of PROM chips required to contain an application program.

Also contained within this table is a user configurable jump address to a user defined non-maskable interrupt (NMI) service routine. The address of this routine must be POKED into the address portion of the jump instruction. When NMI causes a jump to location &H66 , a jump to the user defined NMI routine will be correspondingly executed. A service routine must be provided at that address to handle user required operations.

This Checksum Table furnishes PBASIC with the information necessary to assure that the PROMed application program is equivalent to the RAM version that was generated initially. When the PROM application is RUN, the checksum test is performed and an appropriate message is displayed on checksum failure.

ORGANIZATION OF CHECKSUM TABLE
FIGURE C-1

HEX ADDRESS	TABLE CONTENTS	
8000	CHECKSUM FOR CHIP	1
8001	" " "	2
8002	" " "	3
8003	" " "	4
8004	" " "	5
8005	" " "	6
8006	" " "	7
8007	" " "	8
8008	" " "	9
8009	" " "	10
800A	" " "	11
800B	" " "	12
800C	" " "	13
800D	" " "	14
800E	" " "	15
800F	" " "	16
8010	" " "	17
8011	" " "	18
8012	" " "	19
8013	" " "	20
8014	" " "	21
8015	" " "	22
8017	" " "	23
8018	" " "	24
8019	" " "	25
801A	" " "	26
801B	" " "	27
801C	" " "	28
801D	" " "	29
801E	VARIABLE STORAGE	(LOWER BYTE)
801F	START ADDRESS	(UPPER BYTE)
8020	NUMBER OF CHIPS	(1-29)
8021	TYPE OF CHIP	
8022	JUMP	
8023	NMI Service	(LOWER BYTE)
8024	Routine	(UPPER BYTE)
8025	RESERVED	
8026	START OF SOURCE	

APPENDIX D

MATHEMATICAL FUNCTIONS

D.1 PBASIC MATH VERSIONS

D.1.1 PBASIC SOFTWARE (S/W) MATH VERSION (PBASIC)

This version of PBASIC provides all mathematical operations through software simulation routines. These are efficient software routines to calculate values as follows:

TYPE	PRECISION	RANGE
INTEGER	5	-32768 to 32767
SINGLE PRECISION	7 DIGITS	2.9387 *10**-39 to 1.7014*10**38
DOUBLE PRECISION	16 DIGITS	(SAME AS SINGLE)

D.1.2 PBASIC HARDWARE (H/W) 9511 MATH VERSION (PBASIC/9511)

This version of PBASIC provides the user with a high speed 9511 math processor to increase computational speed. The 9511 provides hardware math functions for the following:

TYPE	PRECISION	RANGE
INTEGER	5	-32768 to 32767
SINGLE PRECISION	7 DIGITS	5.4210*10**-20 to 9.2234*10**18
DOUBLE PRECISION	16 DIGITS	2.9387*10**-39 to 1.7014*10**38

Double precision operations (+, -, *, /) are simulated with software due to hardware limitations of the 9511 math chip. It is important to note that all double precision transcendental functions are converted to single precision format and then processed by the MDX-Math board. Refer to MDX-MATH Operations Manual (MK79741) for detail information of the AM9511.

D.2 DERIVED FUNCTIONS

Functions that are not intrinsic to PBASIC may be calculated as follows.

Function	PBASIC Equivalent
SECANT	SEC(X)=1/COS(X)

COSECANT	`CSC(X)=1/SIN(X)`
COTANGENT	`COT(X)=1/TAN(X)`
INVERSE SINE	`ARCSIN(X)=ATN(X/SQR(-X*X+1))`
INVERSE COSINE	`ARCCOS(X)=-ATN (X/SQR(-X*X+1))+1.5708`
INVERSE SECANT	`ARCSEC(X)=ATN(X/SQR(X*X-1))` `+SGN(SGN(X)-1)*1.5708`
INVERSE COSECANT	`ARCCSC(X)=ATN(X/SQR(X*X-1))` `+(SGN(X)-1)*1.5708`
INVERSE COTANGENT	`ARCCOT(X)=ATN(X)+1.5708`
HYPERBOLIC SINE	`SINH(X)=(EXP(X)-EXP(-X))/2`
HYPERBOLIC COSINE	`COSH(X)=(EXP(X)+EXP(-X))/2`
HYPERBOLIC TANGENT	`TANH(X)=(EXP(X)-EXP(-X))/(EXP(X)+EXP(-X))`
HYPERBOLIC SECANT	`SECH(X)=2/(EXP(X)+EXP(-X))`
HYPERBOLIC COSECANT	`CSCH(X)=2/(EXP(X)-EXP(-X))`
HYPERBOLIC COTANGENT	`COTH(X)=(EXP(X)+EXP(-X))/(EXP(X)-EXP(-X))`
INVERSE HYPERBOLIC SINE	`ARCSINH(X)=LOG(X+SQR(X*X+1))`
INVERSE HYPERBOLIC COSINE	`ARCCOSH(X)=LOG(X+SQR(X*X-1)`
INVERSE HYPERBOLIC TANGENT	`ARCTANH(X)=LOG((1+X)/(1-X))/2`
INVERSE HYPERBOLIC SECANT	`ARCSECH(X)=LOG(1/X+SQR(1/X^2-1))`
INVERSE HYPERBOLIC COSECANT	`ARCCSCH(X)=LOG(1/X+SQR(1/X^2+1))`
INVERSE HYPERBOLIC COTANGENT	`ARCCOTH(X)=LOG((X+1)/(X-1))/2`

APPENDIX E

Converting Programs to PBASIC

If you have programs written in a BASIC other than PBASIC, some minor adjustments may be necessary before running them with PBASIC. Here are some specific items to look for when converting BASIC programs.

E.1 STRING DIMENSIONS

Correct all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the PBASIC statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for PBASIC string concatenation.

In PBASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I thru position J, must be changed as follows:

Other BASICS	PBASIC
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASICS	PBASIC
A$(I)=X$	MID$(A$,I,1)=X$
A$(I,J)=X$	MID$(A$,I,J-I+1)=X$

E.2 MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. PBASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0.

Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

E.3 MULTIPLE STATEMENTS

Some BASICs use a backslash (\) to separate multiple statements on a line. With PBASIC, be sure all statements on a line are separated by a colon (:).

E.4 MAT FUNCTIONS

Programs using the MAT functions available in some PBASICs must be rewritten using FOR...NEXT loops to execute properly.

APPENDIX F

Summary of Error Codes and Error Messages

Error Code	Message
1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	Syntax Error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, no space following operand, etc.).
3	Return without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	Out of DATA A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	Illegal function call A parameter that is out of range is passed to a math or string function. This error may also occur as the result of: a. a negative or unreasonably large subscript b. a negative or zero argument with LOG c. a negative argument to SQR d. a negative mantissa with a non-integer exponent e. a call to a USR function for which the starting address has not yet been given f. an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.
6	Overflow The result of a calculation is too large to be represented in PBASIC'S number format. If underflow occurs, the result is zero and execution continues without an error.
7	Out of memory A program is too large, has too many FOR loops or GOSUBs, too many variables, or

expressions that are too complicated.

8 Undefined line
A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.

9 Subscript out of range
An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

10 Duplicate Definition
Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.

11 Division by zero
A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

12 Illegal direct
A statement that is illegal in direct mode is entered as a direct mode command.

13 Type mismatch
A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

14 Out of string space
String variables exceed the allocated amount of string space. Decrease the size and number of strings, labels, or arrays.

15 String too long
The string was longer than 255 characters.

16 String formula too complex
A string expression is too long or too complex. The expression should be broken into smaller expressions.

17 Can't continue
An attempt is made to continue a program that:
a. has halted due to an error,

b. has been modified during a break in execution, or
c. does not exist.

18 Undefined user function
A USR function is called before the function definition (DEF statement) is executed.

19 No RESUME
An error trapping routine is entered but contains no RESUME statement.

20 RESUME without error
A RESUME statement is encountered before an error trapping routine is entered.

21 Unprintable error
An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

22 Missing operand
An expression contains an operator with no operand following it.

23 Line buffer overflow
An attempt is made to input a line that has too many characters.

26 FOR without NEXT
A FOR was encountered without a matching NEXT.

29 WHILE without WEND
A WHILE statement does not have a matching WEND.

30 WEND without WHILE
A WEND was encountered without a matching WHILE.

<><><><><> AM9511 hardware is expected but is not present <><><><><><>
No AM9511 in system and math overflow is generated on all math functions.

***** ERROR 07 I/O TIME OUT CP0:
Line Printer is off line. Type Control-C to abort.

*****NO RAM MEMORY-OR-BAD FIRST CHIP*****
The memory is incorrectly configured. PBASIC stops execution when this error occurs.

APPENDIX G

Assembly language Subroutines

PBASIC has provisions for interfacing with assembly language subroutines. The USR function allows assembly language subroutines to be called in the same way BASIC's intrinsic functions are called.

G.1 MEMORY ALLOCATION

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). PBASIC uses all memory available from its starting location up, so only the topmost locations in memory should be set aside for user subroutines. The assembly language subroutine must be loaded into memory by PBASIC by means of the POKE statement.

When an assembly language subroutine is called, the stack pointer is set up for 8 levels (16 bytes) of stack storage. If more stack space is needed, PBASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. PBASIC's stack must be restored, however, before returning from the subroutine.

G.2 USR FUNCTION CALLS

In PBASIC, the format of the USR function is

USR[<digit>](argument)

where <digit> is from 0 to 9 and the argument is any numeric or string expression. The parameter <digit> specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement defines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A may be one of the following:

Value in A	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating point number

8 Double precision floating point number

If the argument is a number, the [H,L] register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC-3 contains the lower 8 bits of the argument and
FAC-2 contains the upper 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-3 contains the lowest 8 bits of mantissa and
FAC-2 contains the middle 8 bits of mantissa and
FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).
FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

If the argument is a double precision floating point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the [D,E] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

CAUTION: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

```
A$ = "PBASIC"+""
```

This will copy the string literal into string space and will prevent alteration of program text during a subroutine call.

G.3 CALL STATEMENT

PBASIC user function calls may also be made with the CALL statement. A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return via a simple "RET." (CALL and RET are Z80 instructions, see a Z80 programming manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine.

That parameter is the address of the low byte of the argument.

Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

2. If the number of parameters is greater than 3, they are passed as follows:
   1. Parameter 1 in HL.
   2. Parameter 2 in DE.
   3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for the correct number or type of parameters.

If the subroutine expects more than 3 parameters, and needs to transfer them to a local data area, there is a system subroutine which will perform this transfer. This argument transfer routine is named $AT, and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subroutine is responsible for saving the first two parameters before calling $AT. For example, if a subroutine expects 5 parameters, it should look like:

```
SUBR:   LD      (P1),HL         ;SAVE PARAMETER 1
        EX      DE,HL
        LD      (P2),HL         ;SAVE PARAMETER 2
        LD      A,3             ;NO. OF PARAMETERS LEFT
        LD      HL,P3           ;POINTER TO LOCAL AREA
        CALL    $AT             ;TRANSFER THE OTHER 3 PARAMETERS
        .
        .
        .
        [Body of subroutine]
        .
        .
        .
        RET                     ;RETURN TO CALLER
P1:     DEFS    2               ;SPACE FOR PARAMETER 1
P2:     DEFS    2               ;SPACE FOR PARAMETER 2
P3:     DEFS    6               ;SPACE FOR PARAMETERS 3-5
```

A listing of the argument transfer routine AT follows.

```
00100   ;         ARGUMENT TRANSFER
00200   ;[B,C]  POINTS TO 3RD PARAM.
00300   ;[H,L]  POINTS TO LOCAL STORAGE FOR PARAM 3
00400   ;[A]    CONTAINS THE # OF PARAMS TO XFER (TOTAL-2)
00500
00600
00700           ENTRY   $AT
00800   $AT:    EX      DE,HL           ;SAVE [H,L] IN [D,E]
00900           LD      H,B
01000           LD      L,C             ;[H,L] = PTR TO PARAMS
01100   AT1:    LD      C,(HL)
01200           INC     HL
01300           LD      B,(HL)
01400           INC     HL              ;[B,C] = PARAM ADR
01500           EX      DE,HL           ;[H,L] POINTS TO LOCAL STORAGE
01600           LD      (HL),C
01700           INC     HL
01800           LD      (HL),B
01900           INC     HL              ;STORE PARAM IN LOCAL AREA
02000           EX      DE,HL           ;SINCE GOING BACK TO AT1
02100           DEC     A               ;TRANSFERRED ALL PARAMS?
02200           JR      NZ,AT1          ;NO, COPY MORE
02300           RET                     ;YES, RETURN
```

When accessing parameters in a subroutine, don't forget that they are pointers to the actual arguments passed.

NOTE:

It is entirely up to the programmer to see to it that the arguments in the calling program match in number, type, and length with the parameters expected by the subroutine. This applies to PBASIC subroutines, as well as those written in assembly language.

G.4 INTERRUPTS

Assembly language subroutines can be written to handle interrupts. All interrupt handling routines should save the stack, registers A-L, IX, IY, and the PSW. Interrupts should always be re-enabled before returning from the subroutine, since an interrupt automatically disables all further interrupts once it is received. The user should be aware of which interrupt vectors are free.

APPENDIX H

ASCII Character Codes

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	^
009	HT	052	4	095	_
010	LF	053	5	096	`
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

ASCII codes are in decimal.

LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout