SHEPARDSON
MICROSYSTEMS INCORPORATED

OPTIMIZED SYSTEMS SOFTWARE

OSS BASIC

for the Apple II (R)

OPTIMIZED SYSTEMS SOFTWARE

OSS BASIC

for the Apple II (R)

Feb 1980

Version 1.0

Optimized Systems Software
Shepardson Microsystems, Inc.
20823 Stevens Creek Blvd, Bldg C4-H
Cupertino, CA 95014
Telephone: 408-257-9900

TABLE OF CONTENTS

START-UP 1
- To Start-up . . . 1
- Warm Start . . . 1
- Back-up Copy . . . 1

STATEMENTS AND FUNCTIONs 2
- Syntax Conventions . . . 2
- Types of Items . . . 3
- Execution Control Statements . . . 5
- Miscellaneous Statements . . . 7
- Assignment Statements . . . 9
- Input/Output Statements . . . 10
- Program Control Statements . . . 15
- Arithmetic Functions . . . 16
- String and String Related Functions . . . 17
- Machine Access Functions . . . 18

GENERAL INFORMATION 19
- Key Word . . . 19
- Syntaxing and Internal Format . . . 20
- Deferred and Direct Mode . . . 20
- Statement Save Area . . . 21
- Arrays . . . 21

STRINGS 22
- Substrings . . . 22
- Concatenation . . . 23

OPERATORS 24
- Precedence . . . 24
- And, Or, Not . . . 24

NUMBERS 25

USR FUNCTION 26

INTERFACING TO GRAPHICS 28

ERROR PROCESSING 30
- Syntax Errors . . . 30
- Execution Errors . . . 31
- Error Number Description . . . 31
- Error Trapping . . . 36

NOTES 37

MEMORY MAP 38

SYNTAX SUMMARY 39
- Statements . . . 39
- Functions . . . 40

ERROR SUMMARY 41
- Basic Errors . . . 41
- FMS/OS Errors . . . 42

START - UP

TO START UP:

Put the OSS diskette in disk drive 1. Enter :

6 control-P (return)

This will load the Operating System and execute CP/A. Now enter :

BASIC (return)

This will load the Basic and start executing it. When the

READY

appears on the screen, the user may begin.

WARM START:

If the user goes to the Apple II (R) monitor after start-up, he can return to Basic by entering :

control-Y (return)

This will perserve the user's statements already in memory. It will also leave the current low and high memory addresses unchanged.

The user can return to CP/A using the Bacis command CP. He can then re-enter Basic by using the CP/A command RUN (if he has not loaded another program). This does a warm start.

BACK-UP COPY

To make a back-up copy of Basic on another diskette, use the CP/A SAVE command.

Start address	8000
End address	A800
File name	BASIC.COM

NOTE: For a full explanation of CP/A commands, see the Control Program/Apple documentation.

SYNTAX CONVENTIONS

The following conventions are used in this manual.

1. Capital letters denote keywords, etc., which must be typed by the user exactly as shown (e.g. PRINT, RUN).

2. Lower case letters denote the types of items which may be used. The various types are shown on the next page. (e.g., avar, sexp)

3. Items enclosed in square brackets (e.g., [,var]) are optional.

4. Items enclosed in square brackets together with ellipses imply that the item shown may repeated any number of times. (thus [,exp...] is equivant to [,exp, exp, exp...], etc.).

5. Multiple items in braces indicate that any one may be used. (e.g. {END } implies END and STOP are equivalent statements).
{STOP}

TYPES OF ITEMS

The following types of items are used in decribing the syntax of statements and functions:

GENERAL ITEMS:

avar — Arithmetic VARiable, a storage location for a numeric value. Variable names are 1 to 16 alphanumeric characters and must start with an alphabetic character.

Example: TOTAL , COUNTER3

svar — String VARiable, a storage location for a string of characters (bytes). Same name rules as "avar" except last character must be a dollar sign ("$") which is included in the count of characters in the name.

Example: NAME$, ADDRESS$

mvar — Matrix VARiable, an element of an array (matrix) of numeric values. The name of the matrix is similar to a string variable name except that the last character must be a left parenthesis.

Example: COUNT(NUM) indicates the NUMth element of the array COUNT.

asvar — "avar" or "svar"

var — VARiable, any of "avar", "svar" or "mvar"

aop — The arithmetic operators
+ - * / **

(+ and - can be unary or binary operators)

lop — The logical operators
< <= > >= <> =
AND NOT OR

(NOT is an unary operator)

lexp — Logical Expression, generally composed of "aexp lop aexp" or "sexp lop sexp"; a logical expression evaluates to "true" (represented numerically by a constant 1) or "false" (numerically, 0).

Example: 1<2 is true, "CAT" = "DOG" is false.

sexp — String EXPression, can consist of a string variable, string literal, or a function which returns a string value, No

operators are allowed in a sexp.

Example:
```
ADDRESS$
"SMITH, JOHN"
CHR$(41)
```

aexp — an Arithmetic EXPression, generally composed of "aexp aop aexp" recursively where each aexp element may be an "lexp" "avar", "mvar", numeric literal, or arithmetic function. The arithmetic expression may start with an unary operator.

Example:
```
-A+3
2*SIN(30)
7+INT(B)
A$>B$
```

exp — Either "aexp" or "sexp"

linenum — An expression which is rounded to an integer value. Must be between 1 and 32767.

FILE ITEMS:

fn — file number, an aexp that is rounded to the nearest integer and must be in the range 1 to 7.

filspc — decribed under I/O statements.

mode — an aexp which indicates type of I/O to be performed on a file.

EXECUTION CONTROL STATEMENTS

GOTO linenum

GOTO transfers execution to the line at "linenum" It can also be written GO TO.

GOSUB linenum

GOSUB calls a subroutine which begins at "linenum".

RETURN

RETURN transfers execution to the next statement after the last executed GOSUB.

ON aexp {GOTO } linenum [,linenum...]
{GOSUB}

ON first evaluates aexp and rounds the result to the nearest integer. Then executes a GOTO (or GOSUB) to the 1st linenum in the list if the value is 1, the 2nd if the value is 2, etc. If the value = 0 or is > the number of linenum's in the list, then control falls through to the next line.

IF aexp THEN {statement}
{linenum }

If aexp is "true" (evaluates to non-zero), the statement following the THEN (and any subsequent statements on the line) is/are executed.

If aexp is "false" (evaluates to zero), then control passes to the next sequential line.

The form "THEN linenum" is equivalent to "THEN GOTO linenum".

```
Example:        1000 A=3:B=4:C=3
                2000 IF A=B THEN PRINT "X":PRINT"Y"
                3000 PRINT "Z"
```

Since A=B is false, executing line 2000 will cause control to pass to the next line and will print:
Z
(Y will not be printed.)

```
Example:        1000 A=3:B=4:C=3
                2000 IF A=C THEN PRINT "X": PRINT "Y"
                3000 PRINT "Z"
```

Executing this program will cause the print out:
X
Y
Z

FOR avar = aexp1 TO aexp2 [STEP aexp3]
NEXT avar

When FOR is executed, avar is given the value aexp1.
When NEXT is executed, aexp3 (which takes the value +1 if not given) is added to avar. If avar is then less than or equal to aexp2, control passes to the statement following the FOR, else control falls through to the statement after NEXT.

NOTE: All loops execute at least once.
"aexp3" may be positive or negative.
"avar" is required for NEXT.

MISCELLANEOUS STATEMENTS

POP

POP causes the information about the last GOSUB or FOR to be discarded.

Example:

```
1000 GOSUB 2000
1100 PRINT "HELLO"
1200 END
2000 GOSUB 3000
2100 PRINT "NO WAY"
2200 STOP
3000 PRINT "THIS IS A TEST"
3100 POP
3200 RETURN
```

In this example, the POP at line 3100 will discard information about the last GOSUB (the one at 2000). Then executing the RETURN at 3200 will cause control to pass to line 1100. Executing the program causes this print out:

```
THIS IS A TEST
HELLO
```

DIM svar(aexp)
DIM mvar(aexp)
DIM mvar(aexp,aexp)

DIM lets the user declare the "size" of a string or array.
First, aexp is evaluated and rounded to the nearest integer value.
When the parameter is svar the value is the number of characters in the string.
When the parameter is mvar, the value is the largest subscript. Since the smallest value is 0, there will be value+1 elements in the array.
Arrays may be two dimensional.

NOTE: Multiple strings or arrays may be dimensioned in the same DIM statement.

Example: DIM A(3,2), B$(7), C(3)

This defines a 4 by 3 array whose last element is A(3,2), a string with 7 characters, and an array with 4 elements: C(0), C(1), C(2), C(3).

NOTE: All string and arrays must be DIMensioned before being used. Attempting to access an element or character outside of the size will cause an error. So will attempting to access a string or array that has not

been DIMed.

NOTE:

DIM does not zero array or string elements. The user may do this with a FOR/NEXT loop.

NEW

NEW erases the current program and all references to variables from memory.

CLR

CLR zeros all variables and undimensions all arrays and strings

NOTE: CLR does not zero the memory used by strings and arrays.

LOMEM aexp

The aexp is evaluated and rounded to nearest integer. This value is set as the new LOMEM address (the address at which the user tables start). Then a NEW is done.

NOTE: LOMEM destroys any program currently in memory.

REM ASCII characters

REM denotes a remark or comment. It has no effect on execution of the program. The rest of the line is ignored.

DEG
RAD

These statements only affect the trig functions (SIN, COS, ATN). When DEG is specified, the arguments of SIN and COS and the result of ATN are assumed to be given in degrees. When RAD is specified, these values are assumed to be in radians. The default for Basic is radians.

CP

CP returns control of the system back to the OSS Control Program.

BYE

BYE returns control of the system to the APPLE II(R) monitor.

POKE aexp1, aexp2

"aexp1" and "aexp2" are evaluated and rounded to the nearest integer.
Then the contents of the byte at memory location aexp1 is changed to aexp2.
0 <= aexp1 <= 65535
0 <= aexp2 <= 255

Example: POKE 5000,10

This will cause location 5000 (decimal) to contain 10 (decimal).

NOTE: The function PEEK will allow the user to get a byte from a memory location.

ASSIGNMENT STATEMENTS

[LET] svar = sexp
[LET] {avar} = aexp
{mvar}

LET assigns the value of the expression on the right side of the equal sign to the variable element on the left side.

Example: A$=CHR$(48)
LET B(1,2) = C+3

READ asvar [,asvar...]
DATA ASCII characters [, ASCII characters...]
RESTORE [linenum]

These statements allow data to be stored and retrieved within the program body.
READ takes the next parameter from a DATA statement (ASCII characters up to a comma) and assigns it to "asvar".
If "asvar" is an "avar" then the ASCII characters must represent a numeric value.
If "asvar" is a "svar" then the ASCII characters up to a comma are assigned to the string.

RESTORE indicates the first DATA statement or the DATA statement at linenum. The subsequent READ will access that DATA statement for its parameters.

INPUT Will be decribed in the next section.

INPUT/OUTPUT STATEMENTS

FILE SPEC:

Many I/O statements include a file specification (filspc). This file spec must be a string and have the following format.

filspc - "dev : filename [.file ext]"

dev - {A}
{B}
A indicates a disk in slot 7 drive 1.
B indicates the disk in slot 7 drive 2.
For information on specifications for other devices see Operating System (OS) documentation.

filename
1-8 alphanumeric characters, the first being alpha.

file ext
0-3 alphanumeric characters.

This file spec may be a literal string ("A:ABCD.X") or a string variable which has previously been assigned the proper format (A$ where A$ = "B:ABC.XXX").

FILE NUMBER:

In all I/O statements, #fn is a file number in the range 1 to 7. "fn" is given as an aexp which is rounded to the nearest integer. In statements where #fn is optional and is omitted, the I/O uses the keyboard for input or the screen for output.

STATEMENTS:

OPEN #fn, aexp1, aexp2, filspc

OPEN prepares a file for access and assigns it the file number "fn".

fn - file number [1-7]
aexp1 - I/O mode
4 - input
6 - directory access
8 - output
9 - append
12 - update

aexp2 - device dependent information or 0.

NOTE: After OPENing a file, the file number is used to designate the file in other I/O statements. Two OPEN files cannot have the same file number.

Example: OPEN #2,4,0, "A:DUT.SRC"

This will cause the file DUT.SRC on disk 1 to be opened for input and will associate file number 2 with the file.

CLOSE #fn

CLOSE closes the file associated with the file number "fn".

PRINT [#fn {;}] exp [{,}exp...] {,}
{,} {;} {;}

PRINT puts the ASCII equivalents of the given expressions to the file specified (or the screen). sexp's are simply output (without any conversion --i.e., the full 8-bit byte is output) from their beginning to their length. aexp's are converted to printable form. A comma causes "tabbing" to the next tabular column. A following semicolon causes no spacing.

If the PRINT statement ends in a , or ; then a subsequent PRINT starts outputing at the last location, otherwise the next PRINT will start on a new line.

Example:

```
100 DIM A$(10)
200 A$ = "VALUE ="
300 A = 100
400 PRINT A$,A
500 PRINT A$;A
600 PRINT A$,
700 PRINT A
800 PRINT A$;
900 PRINT A
1000 PRINT A$
1100 PRINT A
2000 END
```

This program causes the following printout:

```
VALUE=          100
VALUE=100
VALUE=          100
VALUE=100
VALUE=
100
```

NOTE: A "," after #fn causes tabbing before first character is printed. A ";" does not cause the tabbing.

INPUT [#fn,] var [,var...]

INPUT requests ASCII input from the specified file number (or the keyboard). It uses a ? prompt. If a "svar" is specified, it accepts a string of characters without transformation until an end-of-line (carriage return) is dectected.
For an "avar" or "mvar", numeric data is converted to internal

form.

NOTE: While numeric data can be terminated by either a comma or a carriage return, string data can only be terminated by a carriage return. So when input is coming from the keyboard and the "svar" is followed by another variable, the user must enter carriage return to end the "svar". Basic will then prompt the user with another ? for the next variable.

```
Example:        1000 DIM A$(50)
                2000 INPUT A$,B
```

When running this program Basic prompts the user with ?. IF the user enters:

```
        JONES, SALLY
```

this becomes the value of A$ and then Basic prompts again for the value of B.

GET `#fn, avar`

GET inputs a single byte from the file specified by "fn" and stores it in avar.

PUT `#fn, aexp`

PUT outputs a single byte (aexp) to the file specified by "fn".

NOTE `#fn, {avar1},{avar2}`
`{mvar1},{mvar2}`

NOTE returns the current sector # and byte within the sector for the file specified by "fn".
Current sector # is assigned to the first variable.
Byte within the sector is assigned to the second variable.

POINT `#fn, {avar1},{avar2}`
`{mvar1},{mvar2}`

POINT sets the current sector # and byte within the sector for the file specified by "fn".
Current sector is set to 1st variable.
Byte within sector is set to 2nd variable.

STATUS `#fn, {avar}`
`{mvar}`

STATUS gets the status of the last I/O performed on device #fn and assigns it to the given variable.

LIST `[filspc]`
LIST `[filspc,] linenum1 [,linenum2]`

LIST lists the program currently in memory to the screen (or to the file specified if "filspc" is given).
If two linenum's are given, only the lines from linenum1 to linenum2 (inclusive) are listed. If a single linenum is given, only that line is listed.

ENTER filspc

ENTER brings a program that was LISTed (ASCII source) back into memory from the specified file. Each statement is syntaxed as it comes in.

SAVE filspc

SAVE puts the program currently in memory to the file specified in an internal format (not in ASCII).

LOAD filspc

LOAD gets a program that has been SAVEd from the file specified. No syntaxing is done since the program is already in internal format.

XIO cmd, #fn, aexp1, aexp2, filspc1

XIO is used to specify various functions to a specific device handler.

Currently this command allows the user to perform various functions on a disk or disk file.
The value of "cmd" tells which function.

cmd	function
32	rename (filspc1 = dev: oldname,newname)
33	delete file
35	lock
36	unlock
37	point
38	note

"fn" tells what file number to associate with the file for this operation.

"aexp1" and "aexp2" are 0 for the functions currently defined.

"filspc" specifies the device and file.

Example: XIO 33,#6,0,0,"A:NOGOOD"

This will cause the file NOGOOD on disk 1 to be deleted. File number 6 will be associated with this file during the process.

NOTE: If the user has added his own device handler (see Operating System (OS) documentation), the XIO cmd can have special meaning to that handler.

PROGRAM CONTROL STATEMENTS

RUN
RUN filspc

RUN without parameters causes the program currently in memory to start execution at the first line number. All variables are set to zero and all strings and arrays are undiminsioned (as if a CLR had been done).

Entering:
```
        RUN     filspc
```
causes two commands to be executed:
```
        LOAD    filspc
        RUN
```

NOTE: In order to LOAD the file it must have be SAVEd (not LISTed).

STOP
END

These statements cause the program to terminate. They are equivalent except that STOP prints out a message giving the current line number.

NOTE: END is not required at the end of a program.

CONT

CONT allows program execution to continue after a STOP or END at the next line number (not next statement).

TRAP linenum

If an error is encountered after a TRAP statement has been executed, control is transfered to the routine at "linenum" This allows the user to have a routine to process errors.

NOTE: TRAP can be disabled by giving a line number greater than 32767.

(More information about error processing can be found in the Error section).

ARITHMETIC FUNCTIONS:

ABS(aexp) Returns absolute value of aexp.

EXP(aexp) Returns exponential of aexp.

Example: A = EXP(X)
A will be assigned the value E^X where E=2.7182818...

FRE(aexp) Returns number of bytes of memory still avaiable.
Aexp is a dummy argument. Its value has no effect.

INT(aexp) Returns the next smaller integer.

Example: INT(3.8) returns 3
INT(-3.8) returns -4

SGN(aexp) Returns an indication of the sign of the argument:

+1 if aexp >0
0 if aexp =0
-1 if aexp <0

SQR(aexp) Returns square root of aexp.

LOG(aexp) Returns natural log of aexp.

CLOG(aexp) Returns common log (base 10) of aexp.

RND(aexp) Returns a pseudo-random number in the range 0 to 1
If aexp is less than 0, the random number generator
is reseeded before a value is returned.

SIN(aexp) Returns sine of aexp.

COS(aexp) Returns cosine of aexp.

ATN(aexp) Returns arc tangent of aexp.

The statements DEG and RAD determine whether
the argument/result is assumed to be in degrees
or radians for the trig functions. The default
is radians.

STRING FUNCTIONS / STRING RELATED FUNCTIONS

LEN(sexp) Returns the current length (not DIM size) in bytes of the string argument sexp.

VAL(sexp) Looks at the leading numerics in the string (up to 255) and returns their value as a number.

```
Example:        A = VAL("57A6B")
```

"A" will be assigned the value 57.

```
Example:        PRINT VAL("ABC")
```

This will cause an error.

ASC(sexp) Returns a number which is the ASCII equivalent of the first character in the string.

```
Example:        PRINT ASC("A")      prints:
                65    (hex 41)

                PRINT ASC("AB")     prints:
                65

                PRINT ASC("57A")    prints:
                53    (hex 35)
```

STR$(aexp) Returns a string that looks like the PRINTed form of the aexp.

```
Example:        B$ = STR$(352)
                has the same effect as
                B$ = "352"
```

CHR$(aexp) Returns a one byte string that contains the value of aexp.

```
Example:        PRINT CHR$(65)      prints:
                A

                PRINT CHR$(53)      prints:
                5

                PRINT CHR$(07)
                rings the bell.
```

NOTE: CHR$ function can be used to insert a byte of binary data into a string. In other words, "aexp" does not have to represent a printable ASCII character.

Example:

```
DIM A$(10)
A$(1)=CHR$(255)
A$(10)=CHR$(13)
```

This will insert hex FF as first character of string A$ and hex OD (carriage return) as the last.

NOTE: There can be only one STR$ and only one CHR$ in a string compare.

Example: A = STR$(1) > STR$(2) produces unpredictable results.

MACHINE ACCESS FUNCTIONS:

PEEK(aexp)

Converts aexp to an integer value by rounding. It returns the contents of the byte in memory at that value. (Memory can be altered by the POKE statement).

Example:

memory location (decimal)	value (decimal)
5000	1
5001	128
5002	25

A = PEEK(5001) assigns value 128 to A

ADR(svar) Returns the address of a string in memory .

USR (aexp1 [,aexp2...])

Evaluates the expressions and rounds to the nearest integer. It pushes the integer values "aexp2" on the CPU stack and puts the number of arguments in the accumulator. It then calls the machine language subroutine at address "aexp1". When Basic regains control, it uses the value in floating point register 0 (FR0) (see memory map address) as the function value.

NOTE: Use with care; this can be a dangerous function if used improperly.

(For a more detailed explaination see section on USR function).

GENERAL INFORMATION

Any given line may contain multiple statements separated by colons.

A program that is executing can be stopped by hitting the escape key (ESC). LIST can also be stopped in this manner.

Since CONTinue causes the program to start executing at the next line (not statement), STOP and END should generally be the last statements on a line.

EXAMPLE:

```
1000 REM TEST PROGRAM
2000 A=1: STOP: PRINT A
3000 END
```

The "PRINT A" in the program can never be executed since CONT entered after the STOP will cause the program to start at line 3000.

Entering a line number and then a carriage return deletes the line.

KEY WORDS

All statement and function names plus TO, STEP, THEN, AND, NOT, and OR are Basic's key words.

For clarity, key words should not be used as a prefix in variable names.

Blanks are significant characters to Basic and key words must be preceeded and followed by blanks any time an ambiguity would result.

During syntaxing the scan is left to right. If Basic is looking for a keyword next, it will stop as soon as it has found one regardless of the next character. Thus

```
FORI = 1 TO 3
```

is valid syntax.

Example:

```
FOR K = ATOB
A = BAND X
```

Cause syntax errors.

```
FOR K = 1TO2
A = B +3
GOTO 1000
1000IFI=3THEN100
```

Are all valid syntax.

SYNTAXING AND INTERNAL FORMAT

Statements are syntaxed on entry. Space for variables is allocated at entry time (not run time). Lines are saved in an internal (tokenized) format.

In this format:

Each variable and/or reserved word in a program will occupy 1 byte.

String constants will occupy string length+1.

Numeric constants will occupy 7 bytes.

NOTE: When memory space is a consideration, space can be saved by changing frequently used constants to variables.

DEFERRED AND DIRECT MODES

There are two modes of execution for statements.

Direct mode - The statement has no line number and is executed as soon as it is syntaxed.

Deferred mode - The statement has a line number. It is syntaxed on entry, stored, and then executed only when the user types RUN.

All of Basic's statements can be executed either in direct or deferred modes. Some have little or no meaning in one mode or the other. Others may have unexpected meanings. For example:

In deferred mode:

6000 RUN	is equivalent to encountering END and then the user typing RUN.
6000 CONT	has no effect.
6000 NEW	is equivalent to encountering END and then the user typing NEW.

In direct mode:

DATA REM END STOP	have no meaning and cause no action

GOTO	starts or restarts program execution at the line number noted in the GOTO statement.
GOSUB	calls the subroutine at the given line number; Upon RETURN control returns to direct mode statements.

STATEMENT SAVE AREA

The statements a user has entered and other needed information is stored starting at a low memory address and growing towards high memory.

This lomem address can be changed with the LOMEM command. This can be used in order to reserve space for a USR function.

The high memory address can be changed using the POKE statement. (See the memory map for the location).

ARRAYS

Arrays are one or two dimemsional. Either dimension may have the value 0 to 32767. (Total space is limited by memory space). Elements are numbered from zero.

Example:

```
DIM A(3)  Allocates 4 elements.
A(0), A(1), A(2), A(3)

DIM B(1,2) Allocates 6 elements.
B(0,0), B(0,1), B(0,2)
B(1,0), B(1,1), B(1,2)
```

An array must be DIMed before it can be used.

STRINGS

A string may have a length of up to 32,767 characters. When a string is DIMensioned, space is allocated for the string and its maximum length is fixed. The characters in a string are numbered from 1 to the DIMensioned maximum.

A string must be DIMensioned before it can be used.

SUBSTRINGS:

A destination string is one that is being assigned to. Any other string is a source string. In

```
X$=Y$
```

X$ is the destination string, Y$ is the source string.

Substrings are defined as follows:

STRING	definition when destination string	definition when source string
S$	the entire string 1 thru DIM value	from 1st thru LEN character
S$(n)	from nth thru DIMth character	from nth thru LENgth character
S$(n,m)	from the nth thru the mth character	from the nth thru the mth character

It is an error if either the first or last specified character (n and m, above) is outside the DIMensioned size. It is an error if the last character position given (explicitly or implicitly) is less than the first character position.

Example:

Assume: DIM A$(10)
A$ = "VWXYZ"

1) PRINT A$(2) prints:
WXYZ

2) PRINT A$(3,4) prints:
XY

3) PRINT A$(5,5) prints:

Z

4) PRINT A$(7)
   is an error because A$ has a length of 5.

CONCATENTATION

Example:

Assume: DIM A$(5), B$(5), C$(15)
A$="12345"
B$="XYZ"

Then concatenation may be performed as follows:

```
LET C$=A$
LET C$(LEN(C$)+1)=B$
```

Now: PRINT C$ will produce

12345XYZ

(This is equivalant to the C$=A$+B$ found in some basics).

OPERATORS

PRECEDENCE

The following order of precedence will be used when evaluating expressions. Operators on the same line have equal precedence.

< <= > => = <>	When used with strings
+ - NOT	unary operators
^	means a number raised to a power
* /	
+ -	When used as binary operators.
< <= > => = <>	When used for arithmetic comparsions.
AND	
OR	

Precedence may be overridden by parentheses.

No binary operator may have both a numeric and string operand.

Example X=A$=C$>B is valid because it is evaluated as X=(A$=C$)>B

X=A$=(C$>B) is invalid

AND, NOT, OR

AND A logical operator which requires both the left and right arguments to be true for the statement to be true.

NOT A logical operator which reverses the truth of the argument following it.

OR A logical operator which requires either the left or right argument to be true for the statement to be true.

NUMBERS

All numbers in Basic are in BCD floating point.

RANGE:

Floating point numbers must be less than 10E+98 and greater than or equal to -10E-98.

INTERNAL FORMAT:

Numbers are represented internally in 6 bytes. There is a 5 byte mantissa containing 10 BCD digits and a one byte exponent.

The most significant bit of the exponent byte gives the sign of the mantissa (0 for postive, 1 for negative). The least significant 7 bits of the exponent byte gives the exponent in excess 64 notation. Internally, the exponent represents powers of 100 (not powers of 10).

Example:	0.02 = 2 * 10^-2 = 2 * 100^-1
exponent=	-1 + 40 = 3F
0.02 =	3F 02 00 00 00 00

The implied decimal point is always to the right of the first byte. An exponent less than hex 40 indicates a number less than 1. An exponent greater than or equal to hex 40 represents a number greater than or equal to 1.

Zero is represented by a zero mantissa and a zero exponent.

USR FUNCTION

The USR function allows the user to execute machine language subroutines. The arguments of the USR function are the address of the subroutine and any parameters the user wishes to pass it (up to 255). The user may return a value in FRO & FRO+1 (see system memory map). In fact, Basic assumes that whatever is in FRO is the return value of the function.

Basic uses the CPU stack to pass the parameters to the subroutine. The count of the number of parameters is passed in the accumulator. The parameters are each 2 byte integers. When the subroutine gets control it must pull the information off the stack.

The first byte on the CPU stack is the MSB of the 1st parameter. The second byte is the LSB of the 1st parameter. The third byte is the MSB of the 2nd parameter, etc...

Example: A=USR(5000,200,300)

The user routine is at the address 5000 (hex 1388).
When the subroutine gets control the stack looks as follows.

Top Of Stack (last on)		
	00	} parm 1
	C8	}
	01	} parm 2
	2C	}

The A register will contain the number of parms (02).

When the user is ready to return, he puts a return value in FRO and FRO+1. FRO is the LSB and FRO+1 is the MSB. Then he executes a RTS.

CAUTION: ALL PARAMETERS THAT BASIC PUT ON THE STACK MUST BE PULLED OFF PRIOR TO THE RTS. The results are totaly unpredictable if this is not done correctly.

When Basic regains control it takes the 2 byte value in FRO as the function's return value.

The user can reserve space for his USR function by using the LOMEM command to increase lomem (the address where Basic starts the tables its needs). The user can also reserve space by using POKE to change the high memory address (see memory map).

The user could also put his 6502 code into a DATA statement, READ the data, put it into a string with CHR$ or POKE and then call the USR function with the address of the string.

Example: The following program will call the Apple II(R) to clear the screen in text mode. This is not the best way of clearing the screen, but it illustrates the use of a string in calling a USR function.

```
1000 DIM A$(20)
2000 FOR I = 1 TO 4
3000 READ X
4000 A$(I) = CHR$(X)
5000 NEXT I
6000 A=USR(ADR(A$(1)))
7000 DATA 32,88,252,96
```

Where the program represented in the DATA statement is:

```
20 58 FC    JSR     $FC58          ; JUMP TO SUBROUTINE
60          RTS                    ; RETURN
```

Example: A better way to clear the screen would be to call the routine directly.

```
100 A=USR(64600)
```

Where 64600 = 65536-936 (FC58 hex).

NOTE: The "A" in these examples is a dummy variable used to take care of the fact that a function expects a return value.

INTERFACING TO GRAPHICS

The Apple II(R) graphics routines are not directly accessible to the Basic user, but they can be used by means of POKE and the USR function. The following are some examples for low resolution graphics. They are not meant to constitute a complete interface.

To set graphics mode:

```
POKE 49232,0     (49232 = hex C050)
POKE 49235,0     (49235 = hex C053)
```

To set text mode:

```
POKE 49233,0     (49233 = hex C051)
```

POKE address for other graphics modes can be found in the APPLE II(R) reference manual.

To plot a point:

Pass the X & Y values to a USR function which will do set-up and call the APPLE II(R) graphics routines.

Example: Basic program required to plot a point at X=5, Y=10.

```
100 LOMEM 8192 : REM SAVE SPACE FOR PGM
200 POKE 34,20 : REM SET WINDOW
300 POKE 49232,0 : REM SET GR MODE
400 POKE 49235,0
500 A=USR(4096,5,10) : REM CALL FUNCTION
```

The USR function calls a routine at 4096 (hex 1000). The "A" is a dummy variable used to take care of the fact that a function expects a return value.

Example: the 6502 code required to interface to plot.

```
                *=$1000          ; ORIGIN AT 1000 HEX
                PHA              ; SAVE COUNT
PLOT
                JSR     $F832    ; CLEAR GRAPHICS AREA
                LDA     #$FF     ; SET COLOR 15
                STA     $30      ; SET IN COLOR REG
                JSR     $FC58    ; CLEAR TEXT AREA
                PLA              ; PULL # OF PARMS
PLOT1                            ; FROM STACK
                CMP     #2       ; SHOULD = 2
                BNE     OOPS     ; IF NOT ERROR
                PLA
                PLA              ; GET PARM 1 (X)
                TAY              ; PUT IN Y FOR PLOT
                PLA
                PLA              ; GET PARM 2 (Y)
```

```
        JSR     $F800       ; GO TO PLOT
        RTS
OOPS
        <HANDLE ERRORS HERE>
```

This routine will plot in color 15. Calling at PLOT initialzes for plotting and plots. Calling at PLOT1 will do subsequent plots. To plot in various colors the routine could be changed to expect the USR function to also pass a color.

NOTE: This routine is meant to serve as an example of what might be done, and is not complete. For example, the user needs to handle errors such as X and Y being out of range.

NOTE: The color must be set after graphics has been initialized.

ERROR PROCESSING

There are two different type of errors in Basic. A syntax error can occur when a statement is entered. An execution error can occur when the statement is actually executed.

SYNTAX ERRORS

A syntax error indicates that the statement just entered is not in proper format. Its occurrence causes the line to be reprinted on the screen with the word ERROR-. The point at which the error was detected (not necessarily the actual error) is marked with an inverse video character.
A syntax error may occur in either direct or deferred mode.

Example 1)

```
1000 A=A$+B
causes:

1000 ERROR-      A=A$+B
                      -
```

2)

```
A=A$+B
causes:

ERROR-  A=A$+B
             -
```

3)

```
1000 FOR I = ATOB STEP C
causes:

1000 ERROR-    FOR I = ATOB STEP C
                                -
```

Where underscore(-) indicates the character in inverse video.

NOTE: In example 3 the error may be that the user meant to type

```
1000 FOR I = A TO B STEP C
```

but since ATOB is a valid variable name, no error is detected until STEP.

EXECUTION ERRORS

If an error is detected while a statement is being executed, Basic prints out:

ERROR- XX

If the statement had a line number (ie. it was part of a program), Bacis also prints:

AT LINE nnnn

XX represents an error number in the range 1 to 255.
nnnn represents the line number on which the error occured.

So the two forms are:

ERROR- XX
ERROR- XX AT LINE nnnn

The following section is a decription of errors represented by the error number.

ERROR NUMBER DECRIPTION

2 - MEMORY FULL

All avaiable memory has been used. No more statements can be entered and no more variables (arithmetic, string or array) can be defined.

3 - VALUE ERROR

An expression or variable evaluates to an incorrect value.

Example: An expression that can be converted to a two byte integer in the range 0 to 65235 (hex FFFF) is called for and the given expression is either too large or negative.

```
A = PEEK(-1)
DIM B(70000)
```

Both these statments will produce a value error

Example: An expression that can be converted to a one byte integer in the range 0 to 255 hex(FF) is called for and the given expression is too large.

```
POKE 5000,750
```

This statement produces a value error.

Example: A=SQR(-4) Produces a value error.

4 - VARIABLE TABLE FULL

No more variables can be defined. The maximum number of variables is 128.

5 - STRING LENGTH ERROR

A character beyond the DIMensioned or current length of a string has been accessed.

Example:
```
1000 DIM A$(3)
2000 A$(5) = "A"
```

This will produce a string length error at line 2000 when the program is RUN.

6 - READ OUT OF DATA

A READ statement is executed but we are already at the end of the last DATA statement.

7 - LINE NUMBER TOO BIG

A line number larger than 32767 was entered.

8 - INPUT/READ STATEMENT ERROR

The INPUT or READ statement did not recieve the type of data it expected.

Example:
```
INPUT A
```

If the data entered is 12AB then this error will result.

Example:
```
1000 READ A
2000 PRINT A
3000 END
4000 DATA 12AB
```

Running this program will produce this error.

9 - ARRAY/STRING DIM ERROR

Example: A string or an array was used before it was DIMinsioned.

Example: A previously DIMensioned string or array is DIMensioned again.

```
1000 DIM A(10)
2000 DIM A(10)
```

This program produces a DIM error.

10 - EXPRESSION TOO COMPLEX

An expression is too complex for Basic to handle. The solution is to break the calculation into two or

more Basic statements.

11 - FLOATING POINT OVERFLOW/UNDERFLOW

The floating point routines have produced a number that is either too large or too small.

12 - LINE NOT FOUND

The line number required for a GOTO or GOSUB does not exist.
The GOTO may be implied as in:

```
1000 IF A=B THEN 500
```

The GOTO/GOSUB may be part of an ON statement.

13 - NO MATCHING FOR

A NEXT was encountered but there is no information about a FOR with the same variable.

Example:

```
1000 DIM A(10)
2000 REM FILL THE ARRAY
3000 FOR I = 0 TO 10
4000 A(I) = I
5000 NEXT I
6000 REM PRINT THE ARRAY
7000 FOR K = 0 TO 10
8000 PRINT A(K)
9000 NEXT I
10000 END
```

Running this program will cause the following output:

```
0

ERROR- 13 AT LINE 9000
```

NOTE: Improper use of POP could cause this error.

14 - LINE TOO LONG

The line just entered is longer than Basic can handle. The solution is to break the line into multiple lines by putting fewer statements on a line, or by evaluating the expression in multiple statements.

15 - GOSUB/FOR LINE DELETED

The line containing the GOSUB or FOR was deleted after it was executed but before the RETURN or NEXT was executed.
This can happen if, while running a program, a STOP is executed after the GOSUB or FOR, then the line containing the GOSUB or FOR is deleted, then the user types CONT and the program tries to execute the RETURN or NEXT.

Example:
```
1000 GOSUB 2000
1100 PRINT "RETURNED FROM SUB"
1200 END
2000 PRINT "GOT TO SUB"
2100 STOP
2200 RETURN
```

If this program is run the print out is:

```
GOT TO SUB

STOPPED     AT LINE 2100
```

Now if the user deletes line 1000 and then types CONT we get

```
ERROR- 15 AT LINE 2200
```

16 - BAD RETURN

A RETURN was encountered but we have no information about a GOSUB.

Example:
```
1000 PRINT "THIS IS A TEST"
2000 RETURN
```

If this program is run the print out is:

```
THIS IS A TEST

ERROR- 16 AT LINE 2000
```

NOTE: improper use of POP could also cause this error.

17 - EXECUTION OF GARBAGE

If when entering a program line a syntax error occurs, the line is saved with an indication that it is in error. If the program is run without this line being corrected, execution of the line will cause this error.

NOTE: The saving of a line that contains a syntax error can be useful when LISTing and ENTERing programs.

18 - STRING DOES NOT START WITH A VALID NUMBER

If when executing the VAL function, the string argument does not start with a number, this message number is generated.

Example: A = VAL("ABC") produces this error.

19 - LOAD PROGRAM TOO BIG

The program that the user is trying to LOAD is larger than available memory.

This could happen if the user had used LOMEM to change the address at which Basic tables start, or if he is LOADing on a machine with less memory than the one on which the program was SAVEd.

20 - INVALID DEVICE/FILE NUMBER

If the device/file number given in an I/O statement is 0 or greater than 7, then this error is issued.

Example:
```
GET #8,A
PUT #0, B+7
```

Both of these statements will produce this error.

21 - NOT A LOAD FILE

This error results if the user tries to LOAD a file that was not created by SAVE.

ERROR TRAPPING

The TRAP statement allows the user to specify the line number of a routine that will be executed when an error is encountered.

Basic saves the error number and the line number where the error occured for the user. (See memory map for locations). By doing a PEEK the user can see what the error was and determine what action he wishes to take. For example if the error was end-of-file, he may wish to close the file and end the program.

The user can return control to the first line (not statement) after the statement that caused the error by doing a CONT.

The user can also return to the first statement of the line that caused the error. To do this he must use PEEK to get the line number then construct a variable containing that line number.

Example: Assume the memory map says that the line number is at location X (in decimal). The error routine can return control to the 1st statement of the line causing the error by saying:

```
1000 GOTO PEEK(X)+PEEK(X+1)*256
```

NOTE: The TRAP statement must be executed before the error occures.

NOTE: TRAP may be disabled by using a line number greater than 32768.

```
TRAP 40000
```

NOTE: To prevent infinite looping, TRAP is disabled after an error. If the user wants an error routine to be called on the next error, he must re-execute TRAP.

NOTES

DEFAULT FILE NUMBER:

Some Basic commands do not specify a file number but request I/O to or from a device other than the screen or keyboard (SAVE, ENTER, etc). Basic must have a file number to do this, and it uses file number 7. The user should not have this file number assigned to a device/file when this type of command is issued.

ENTERING FROM A PROGRAM

ENTER can be executed from a program. This causes the statements in the file to be merged with the statements in memory. The ENTERed statements are not automatically executed. If the user wishes to ENTER and then execute, he may append a GOTO statement (without a line number) to his file. (See Disk File Manager documentation).

LOMEM/HIMEM

A default low memory address is set when the system is booted up. Basic does NOT automatically reset this value. If a program (for example, a device handler), sets lomem and then BASIC is entered, this address remains unchanged.

Basic does set a default himem which can be changed by POKE.

MEMORY MAP

The following are memory locations used by OSS. For locations used by Apple II(R) monitor, see Apple II(R) Reference Manual.

HEX ADDRESS	USED FOR
0000-001F	Reserved for user
0020-004F	Apple II(R) monitor
0050-005F	Reserved for user
0060-007F	OSS Operating System
0080-00D3	Basic
00D4-00FF	Floating Point work area
0100-01FF	6502 stack
0200-02FF	Apple II(R) input buffer and Basic syntax stack
0300-037F	Basic line buffer
03F0-03F4	Autostart Rom

Specific locations that may be of interest:

BA-BB	Stop line number (after STOP, END or error)
C3	Error number
D4-D9	Floating Point register zero (FR0)
8000	Basic cold start address
8003	Basic warm start address
9FFC-9FFD	Pointer to test for escape key
BFF6-BFF7	Pointer to low memory address
BFF8-BFF9	Pointer to high memory address

SYNTAX SUMMARY

STATEMENTS

```
BYE
CLOSE    #fn
CLR
CONT
CP
DATA     ASCII characters [,ASCII characters....]
DEG
DIM      svar (aexp)
DIM      mvar (aexp)
DIM      mvar (aexp, aexp)
END
ENTER    filspc
FOR      avar = aexp1 TO aexp2 [STEP aexp3]
GET      #fn, avar
GOSUB    linenum
IF       aexp THEN {statement}
                   {linenum  }
INPUT    [#fn ,] var [,var...]
[LET]    svar = sexp
[LET]    {avar} = aexp
         {mvar}
LIST     [filspc]
LIST     [filspc,] linenum [,linenum]
LOAD     filspc
LOMEM    address
NEW
NEXT     avar
NOTE     #fn, {avar}, {avar}
              {mvar}  {mvar}
ON       aexp {GOTO } linenum [,linenum...]
              {GOSUB}
OPEN     #fn, mode, aexp, filspc
POINT    #fn, {avar}, {avar}
              {mvar}, {mvar}
POKE     address, aexp
POP
PRINT    [#fn {;}] exp [{,} exp...] {,}
              {,}        {;}          {;}
PUT      #fn, aexp
RAD
READ     asvar [, asvar...]
REM      ASCII characters
RESTORE  [linenum]
RETURN
RUN      [filspc]
SAVE     filspc
STATUS   #fn, {avar}
              {mvar}
STOP
TRAP     linenum
XIO      cmd, #fn, aexp, aexp, filspc, [,filspc]
```

FUNCTIONS

ABS(aexp)

ADR(svar)

ASC(sexp)

ATN(aexp)

CHR$(aexp)

CLOG(aexp)

COS(aexp)

EXP(aexp)

FRE(aexp)

INT(aexp)

LEN(sexp)

LOG(aexp)

PEEK(address)

RND(aexp)

SGN(aexp)

SIN(aexp)

STR$(sexp)

SQR(aexp)

USR(address [,parameters...])

VAL(sexp)

ERROR SUMMARY

A more detailed explaination of the Basic errors can be found in the section on ERROR PROCESSING.

BASIC ERRORS

2 - MEMORY FULL
3 - VALUE ERROR
4 - VARIABLE TABLE FULL
5 - STRING LENGTH ERROR
6 - READ OUT OF DATA
7 - LINE NUMBER TOO BIG
8 - INPUT/READ STATEMENT ERROR
9 - ARRAY/STRING DIM ERROR
10 - EXPRESSION TOO COMPLEX
11 - FLOATING POINT OVERFLOW/UNDERFLOW
12 - LINE NOT FOUND
13 - NO MATCHING FOR
14 - LINE TOO LONG
15 - GOSUB/FOR LINE DELETED
16 - BAD RETURN
17 - EXECUTION OF GARBAGE
18 - STRING DOES NOT START WITH VALID NUMBER
19 - LOAD PROGRAM TOO BIG
20 - INVALID DEVICE/FILE NUMBER
21 - NOT A LOAD FILE

For the user convenience a summary of the error messages that can be generated by DFM/OS and passed to Basic are included.

DFM/OS ERRORS:

DEC	HEX		MESSAGE
129	(81)	-	DEVICE NOT READY
130	(82)	-	NON EXISTENT DEVICE
131	(83)	-	DATA ERROR
132	(84)	-	INVALID COMMAND
133	(85)	-	DEVICE OR FILE NOT OPEN
134	(86)	-	INVALID IOCB NUMBER
135	(87)	-	WRITE PROTECT
136	(88)	-	END OF FILE
160	(A0)	-	DRIVE # ERROR
161	(A1)	-	TOO MANY OPEN FILES (NO SECTOR BUFFER AVAIABLE)
162	(A2)	-	MEDIUM FULL (NO FREE SECTORS)
163	(A3)	-	FATAL SYSTEM DATA I/O ERROR
164	(A4)	-	FILE # MISMATCH
165	(A5)	-	FILE NAME ERROR
166	(A6)	-	POINT DATA LENGTH ERROR
167	(A7)	-	FILE PROTECTED
168	(A8)	-	COMMAND INVALID (SPECIAL OPERATION CODE)
169	(A9)	-	DIRECTORY FULL
170	(AA)	-	FILE NOT FOUND
171	(AB)	-	POINT INVALID