CMSbasic

This version of CMSbasic is a rewrite and modifications to the orginal BxBasic package written by Steve Arbayo, Copyright:(c) sarbayo, 2001-2011, Bxbasic (aka:Blunt Axe Basic). I have left all the copyrights, etc, in the source code.

Steve has done an outstanding job, I hope my changes do not impact the great work that he has done.

Tom Chandler
2018 - 2020

Table of Contents

Welcome to CMSbasic:

CMSbasic is a Console Mode 32 bit Scripting Engine and Byte Code Compiler.

The Bxbasic dialect is a subset of the GWBasic, QBasic and QuickBasic_4.5 dialects.

How to use CMSbasic:

The **CMSBasic** components are:

CMSBasic

A stand-alone scripting engine (interpreter). With the scripting engine, CMSBasic scripts can be executed and tested. A text scripted program, written in the CMSBasic subset of the Basic dialect, using any text editor may be run.

General Information

Note: CMSbasic is case sensitive. All statement keywords must appear in all upper-case, Such as: LET, DIM, FOR, etc.

All function names must appear in upper-case. Additionally, there are several string, algebraic and device functions.

CMSBasic is executed via CMS under VM370:

```
        run CMSBASIC ( bx1 basic
```

or

        Follow the INSTALL.txt instructions to make an executable
        version of CMSBASIC.

Basic programs are saved in the CMS using the extension of basic.

Keyword's

```
Keyword:            Description:
============        ====================================
```

ABS()

```
Absolute value of expression.
ex:   integer = ABS(integer)
```

ACOS()

```
Calculates the arc cosine of number.
```

ASC()

```
Returns the ascii code of the first character in
a string.
ex:       integer = ASC(a$)
```

ASIN(num)

```
calculates the arc sine of number
usage:
      MyDouble# = ASIN(double#)
      Print "ASIN("; double#; ")="; MyDouble#
```

ATN()

```
Return the arc tangent of a number.
ex:       float# = ATN(float#)
```

ATAN2(n1,n2)

```
calculates the arc tangent of n1/n2
usage:
      MyDouble# = ATAN2(aDouble#, bDouble#)
      Print "ATAN2("; aDouble#; ","; bDouble#;
            ")="; MyDouble#
```

CEIL(num)

```
calculates the smallest whole number that is not
less than number
usage:
      MyDouble# = CEIL(double#)
      Print "CEIL("; double#; ")="; MyDouble#
```

CDBL()

Converts a number to double precision.

```
usage:
      double# = CDBL(Integer)
example:
 Dim intVal = 10
 Dim dblVal# = 0
 dblVal# = CDBL(intVal)
 Print "Double="; dblVal#
```

CHR$()

Returns the character corresponding to the ASCII value of the integer.

```
ex:      string$ = CHR$(integer)
```

CINT()

converts a floating point number to an integer by rounding up the fractional portion of the number.

```
usage:
      result = CINT(float)
example:
Dim float# = 14.9999
Dim intVal = 0
intVal = CINT(float#)

The value of intVal is: 15
```

CLEAR

Sets numeric variables to zero, strings to NULL.

```
Ex:  CLEAR (clears all numeric and string
       variables)
     CLEAR a$, name$, integer%
```

CLNG()

converts a floating point number to a long integer by rounding up the fractional portion of the number.

```
usage:
long% = CLNG(float#)

example:
```

```
Dim lngVal% = 0
Dim dblVal# = 32767.55
lngVal% = CLNG(dblVal#)
Print "long="; lngVal%
```

CLOCK()

Number of clock_ticks since program start.
usage:

```
      number = CLOCK()
```

CLOSE

Closes access to a disk file. I
ex:

```
CLOSE #1
```

COS()

Returns the cosine of a floating point number.
ex:

```
float# = COS(float#)
```

COSH(num)

return hyperbolic cosine of number
usage:

```
      MyDouble# = COSH(double#)
      Print "COSH("; double#; ")="; MyDouble#
```

CSNG()

converts number to single precision, rounding the number when converting it to single precision.
usage:

```
      single! = CSNG(Float#)
```

example:

```
Dim snglVal! = 0
Dim dblVal# = .1453885509
snglVal! = CSNG(dblVal#)
Print "Single="; snglVal!
```

CVD()

Converts an 8 byte string to a double precision number. Used to restore numeric data to numeric form after a disk read.
ex:

```
float# = CVD(string)
```

CVI()

Converts a 4 byte string to an integer number. Used to restore   numeric data to numeric form after a disk read.

ex: integer% = CVI(string)

CVS()

Converts an 8 byte string to a Single precision number. Used to restore numeric data to numeric form after a disk read.
ex: float! = CVS(string)

DATE$

This keyword returns the current date in the format of MM/DD/YYYY
ex: d$ = DATE$

DECLARE

create a SUB function/routine.

DEFAULT

switch/case.

DIM

Declares and dimensions a multi-dimensional string array. In this release, DIM supports string arrays only.

In this example:
```
   DIM string$(3,3,10)
```

the array is an array of 90 string pointers: 3 x 3 x 10 = 90 That is, an array of 90 address pointers and not 90 characters. The above could be best seen as 3 groups (or tablets), each containing 3 pages, with each page containing 10 rows for data (or strings). Each array string (or row) is dynamically created and populated when it is assigned to.

In this example:
```
   DIM string$(10)
```

an array of 10 string address pointers is created. Each of the 10 strings can be of any desired (within reason) length.

The following example dimensions an array and then populates three array pointers:

```
DIM string$(3,3,10)
```

```
string$(1,1,1) = "hello world!"
string$(2,2,2) = "This is a test "
string$(2,2,3) = "of the emergency broadcast"
```

DO

conditional loop.

ELSE

default action.

ELSEIF

alternate condition.

END

Terminates program execution. Every program must have at lease one END/STOP/SYSTEM statement. The END command need not be the last line in the program. ie: it can be anywhere within the program.

```
     IF x = 0 THEN
        GOTO TheEnd
     ENDIF
     ...
TheEnd:
     END
```

END or STOP or SYSTEM can be used interchangeably.

ENDIF

condition terminator.

ENDSUB

declares the end of a SUB function.

EOF()

Returns a Boolean TRUE or FALSE based on whether or not the end-of-file has been reached. Is used in an "IF/ELSE" construct.

```
ex:       IF EOF(1) THEN
              GOTO CloseFile
          ENDIF
```

ERASE

erase an array.

EXP()

returns the natural exponent of number
usage:

```
      MyDouble# = EXP(double#)
      Print "exponent="; MyDouble#
```

FABS(num)

returns the absolute value of a floating point number
usage:

```
      MyDouble# = FABS(double#)
      Print "FABS("; double#; ")="; MyDouble#
```

FIELD

Divides a random access file buffer into fields so that data can be written to disk or read from disk into memory. Each field is identified by a string variable name and is of a specified length. The length must be an integer constant.

```
ex:       FIELD #1, 10 AS firstname$,
               10 AS lastname$, 1 AS initial$
```

FLOOR(num)

returns a double precision, largest whole number, less than number
usage:

```
      MyDouble# = FLOOR(double#)
      Print "FLOOR("; double#; ")="; MyDouble#
```

FMOD(n1,n2)

calculates the floating point remainder of n1 divided by n2
usage:

```
      MyDouble# = FMOD(aDouble#, bDouble#)
      Print "FMOD("; aDouble#; ","; bDouble#;
            ")="; MyDouble#
```

FOR/NEXT

FOR variable = initial value TO final value STEP increment NEXT variable

Creates a program loop that will be executed for a predetermined number of cycles. The value of 'variable' is set to that of 'initial value'. The value of 'variable' is then incremented after each cycle by the value of 'increment'.

In this example, the loop will execute for 15 cycles:

```
   ex:    FOR x = 1 TO 30 STEP 2
          ' do stuff
          NEXT
```

loop construct (includes: [TO][STEP]).

FREXP()

returns the mantissa of a floating point number (n1), as a normalized fraction. The power of 2 exponent of n1 is stored in n2.

```
usage:
MyDouble# = FREXP(aDouble#, Myint)
Print "FREXP("; aDouble#; ","; Myint; ")=";
      MyDouble#
Print "Myint="; Myint
```

GET

Reads a record from a random access disk file and places it in the specified buffer. Record is an integer value. The default record if none is specified is the next logical record.

```
   ex:    GET #1,ndx
```

GOSUB/RETURN

```
 GOSUB label
RETURN
```

Branches to the subroutine beginning at "label". Every subroutine must end with a RETURN command.

```
   GOSUB MyRoutine
   ...
   ...
MyRoutine:
   'do stuff
   RETURN
```

GOTO

Causes an absolute jump to the program line that follows "label". There is no returning from a GOTO.

```
GOTO MyLabel
...
...
MyLabel:
    'do stuff
```

HYPOT(n1,n2)

calculates the length of the hypotenuse of a right triangle, with sides n1 and n2

usage:

```
MyDouble# = HYPOT(aDouble#, bDouble#)
Print "HYPOT("; aDouble#; ","; bDouble#;
")="; MyDouble#
```

IF/ELSEIF/ELSE/ENDIF

```
IF expression THEN
       statement
ELSEIF expression THEN
       statement
ELSE
       default
ENDIF
```

IF/ELSE is a conditional expression construct, where any number of variables and conditions can be tested for logical and boolean values. A simple expression might be:

```
IF x = 0 THEN
       PRINT "x = 0"
ENDIF
```

A more complex expression might look like:

```
IF x >= 0 AND x <= 10 THEN
       PRINT "x=0 to 10",x
ELSEIF x >= 11 AND x <= 20 THEN
       PRINT "x=11 to 20",x
ELSEIF x >= 21 AND x <= 30 THEN
       PRINT "x=21 to 30",x
ELSE
       PRINT "x != 0 to 30",x
ENDIF
```

After the initial IF there may be any number of ELSEIF's, but, there may be only one (1) ELSE and there must always be one(1) (and only one) ENDIF. Additionally, each IF and ELSEIF line must be terminated by a THEN. Expressions may be compounded by the use of AND and OR:

```
IF a$ = b$ AND b$ <> c$ OR c$ = d$ THEN
```

Logical operators are:

=	equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
<>	not equal to conditional expression.

INPUT

Accepts keyboard input and puts it into any number of and type of variable. If INPUT is followed by a semi-colon, as shown here:

```
INPUT ; a$
```

the return key will not be echoed to the screen when it is pressed at the of the input'd data. INPUT may also be followed by a 'prompt' string, which will be displayed in front of the input area:

```
INPUT ; "Enter your firstname:"; fname$
```

Multiple prompts and variables may be entered on the same INPUT line:

```
INPUT ; "Name"; name$, "Age:"; age%, "ID:"; IDno%
```

INPUT$()

Accepts data from a sequential access text file and stores the data in variable(s). Buffer is the file buffer number.

```
INPUT #1, string$, integer%, float!, double#
```

INT()

Converts number into the largest integer value that is less then or equal to number.

```
integer% = INT(float#)
```

LCASE$()

convert upper-case string to lower-case.
usage:

```
      a$ = LCASE$(str$)
      a$ = LCASE$("UPPERCASE TEXT")
```

LDEXP(n1,n2)

calculates n1 times 2, raised to the power of n2
usage:

```
      MyDouble# = LDEXP(aDouble#, Myint)
      Print "LDEXP("; aDouble#; ","; Myint; ")=";
                  MyDouble#
```

LEFT$()

A string slicing function. Returns the specified number of characters from the left portion of string. Number must be in the range of: 1 to 255.

```
   string$ = "hello world!"
   A$ = LEFT$(string$, 5)
```

LEN()

Returns the number of characters in an ascii zero-terminated string.

```
   Integer = length of string$.
```

LET

Assigns the value of expression to variable. The LET keyword is optional. By default, unless indicated otherwise, program lines are assumed to be LET statements. Therefore, the use of LET is optional. Assignments may be of any data type. For this reason, variable names can not be the same as keyword names.

```
   LET integer% = value
   integer% = value
```

CMSBasic is a strongly typed language. With only one exception, all variable names must include a data type specifier, appended to the name. Ordinary integer variables may optionally use the integer data type symbol: %. Any variable name encountered without a data type specifier is assumed to be a simple integer.

```
variable  = integerValue
variable% = integerValue
```

The data type specifiers are:

%	integer
!	float
#	double float
@	long integer
$	string

String assignments can be string to string ssignments, string plus string and string to array, array to string assignments.

```
A$ = "quoted text string"
B$ = A$
C$ = A$ + B$
array$(1,1,1) = A$
data$ = array$(1,1,2)
```

Numeric value assignments may be constants:

```
integer = 123456
double# = 123.456789
```

or, variable data:

```
integer = varname
```

or, complex algebraic expressions:

```
double# = (abc * (xyz + width / 3))
```

LINE INPUT

Inputs an entire line of text into a single string variable, up to 255 characters. LINE INPUT may also be followed by a 'prompt' string, which will be displayed in front of the input area:

```
LINE INPUT ; "Enter your firstname:"; fname$
```

Input is terminated by pressing the enter key.

LINE INPUT#

Inputs an entire line of text from a file, into a single string variable, up to 255 characters.

```
    LINE INPUT #1, A$
```

Input ends when either 255 characters is read in or a newline character is encountered.

LOC()

Returns the current record position within a random access file. Buffer is the file buffer number assigned to the file when it was opened.

```
   integer% = LOC(1)
```

LOF()

Returns the length of the previously opened file in bytes.

```
   integer% = LOF(1)
```

LOG()

returns the natural logarithm of number.
usage:

```
      number = LOG(variable)
      number = LOG(3.14)
```

LOG10()

returns the logarithm base 10 of number.
usage:

```
      number = LOG10(variable)
      number = LOG10(99)
```

LSET

Moves data to the random access buffer and places it In the field name in preparationn for a PUT command. Field name is a string variable defined in a FIELD statement. A file buffer must be FIELD'd before using LSET.

```
   LSET A$ = "text string"
   LSET B$ = string$
   LSET C$ = array$(a,b,c)
```

LSET left-adjusted the data to the left side of the field. IF the data that is being LSET is smaller than the data buffer field, then the empty space in the buffer field will be padded with blank spaces. If the data is larger than the buffer field, then the extra characters will be truncated (removed).

MID$()

Returns the specified number of characters from any portion of a character string. Start is the starting point, counting from the left side. Length is the number of characters to copy. Both number must be in the range of: 1 to 255 and less than the total length of string.

```
string$ = "hello world!"
a$ = MID$(string$, 7, 5)
```

MKD$()

Used in combination with the LSET command, to copy a double precision floating point value into a random access string buffer. The value is stored in it's binary form as an 8-byte string. This is the inverse of the CVD function.

```
LSET a$ = MKD$(double#)
```

MKI$()

Used in combination with the LSET command, to copy an integer value into a random access string buffer. The value is stored in it's binary form as a 4-byte string for a regular integer and an 8-byte string for a long integer. This is the inverse of the CVI function.

```
LSET A$ = MKI$(integer%)
```

MKS$()

Used in combination with the LSET command, to copy a single precision floating point value into a random access string buffer. The value is stored in it's binary form as an 8-byte string. This is the inverse of the CVS function.

```
LSET a$ = MKS$(float!)
```

MODF()

```
MODF(n1,n2):    returns the fractional part of
                n1, the whole part is stored in
                n2
usage:
 MyD# = MODF(aD#, bDouble#)
 Print "MODF("; aD#; ","; bDouble#; ")="; MyD#
Print "bDouble# ="; bDouble#
```

NEXT

See: FOR/NEXT

OPEN

```
OPEN mode, #buffer, path (sequential
                              access file)
OPEN mode, #buffer, path, record size (random access)
```

Creates an input/output path for a disk file or device.

```
Mode is:  "O"    output sequential access (text)
          "I"    input  sequential access (text)
          "A"    append sequential access (text)
          "R"    random random access (binary)
```

Buffer is: the I/O buffer number and file identifier (handle) to be used for accessing this file. Buffer numbers may be in the range of 1 to 99. Buffer numbers need not be used in any particular order. On startup, CMSBasic creates 99 I/O buffer pointers. The '#' symbol IS required.

Path is: the logical path or file name to the device or disk file.

Record size is: (random access only) the number of bytes in the I/O buffer to write to, or read from, the disk file. Record size should always be specified. If record size is not specified, the default record size is set at 256 bytes.

Sequential Access:

```
    OPEN "O", #1, "My.txt"
```

Creates a new file for output. If My.txt already exists, it will be deleted and overwritten.

```
    OPEN "I", #99, filename$
```

Opens an existing file, for input.

```
    OPEN "A", #3, "existing.fil"
```

Creates a new file or opens existing file for output. If file exists, new data written to the file will be appended to the end of the file, leaving pre-existing information intact.

```
    OPEN "O", #1, "LPT1"
```

Opens an Output channel to the printer port: LPT1:

Random Access:

```
    OPEN "R", #7, "MyData.fil", 30
```

Opens and existing or creates a new file, as the case may be, for both read and write access. Each read or write action will I/O 30 bytes.

See: FIELD, LSET, PUT, GET.

POW()

calculates n1 raised to the n2 power.

POW10(num)

calculates 10 raised to the power of num
usage:

```
      MyDouble# = POW10(aDouble#)
      Print "POW10("; aDouble#; ")="; MyDouble#
```

PRINT

Prints numeric or string data to the console screen. A single PRINT command such as:

```
   PRINT
```

will print a newline on the screen. A delimiter or separator must be used between data items. Delimiters may either be a comma ','or a semi-colon ';'. Using a comma after a data item will send a TAB character to the screen:

```
   PRINT "hello",
```

Using a semi-colon after or between items will cause the next item to be printed beside the prior:

```
   PRINT "hello"; "world!"
```

Special ascii characters and non-alpha-numeric characters may be printed by using the CHR$() function:

```
   PRINT CHR$(34); "hello "; world$; CHR$(34)
```

the resulting display will read: "hello world"

```
   ascii 34----^           ^----ascii 34
```

PRINT#

```
PRINT #buffer, data
```

Write unformatted text to a sequential access file. It will appear on disk much as it looks on the screen.

```
   PRINT #1, FName$, LName$, Age%
```

PUT

Writes a record to a random access disk file from the specified buffer. Record is an integer value. The default record if none is specified is the next logical record.

```
   PUT #1,ndx
```

RAND()

Returns a random number, once the random number generator has been seeded.

```
   seed@ = CLOCK()
   RANDOM seed@
   randomnumber@ = RAND()
```

RANDOM

Reseeds the random number generator. Should always be used before using the RAND function. Seed may be any type of number, integer or floating point.

```
   seed% = CLOCK()
```

```
RANDOM seed%
RANDOM 123.4567
```

READ#

Same as INPUT#. Accepts data from a sequential access text file and stores the data in variable(s). Buffer is the file buffer number.

```
READ #1, string$, integer%, float!, double#
```

REM

A line comment. An apostrophy(') my be used instead of the keyword REM.

```
REM    this is a comment and is ignored by the
       compiler
'      this is a comment and is ignored by the
       compiler
```

RETURN

return from subroutine.

RIGHT$()

Returns the specified number of characters from the right portion of a character string. Beginning at the right side, number is the number of characters to copy. Number must be in the range of:

1 to 255 and less than the total length of string.

```
string$ = "hello world!"
a$ = RIGHT$(string$, 6)
```

RSET

Moves data to the random access buffer and places it in the field name in preparation for a PUT command. Field name is a string variable defined in a FIELD statement. A file buffer must be FIELD'd before using RSET.

```
RSET A$ = "text string"
RSET B$ = string$
RSET C$ = array$(a,b,c)
```

RSET right-adjusts the data to the right side of the field. IF the data that is being RSET is smaller than the data buffer field, then the empty space in the buffer field will be padded with blank spaces. If the data is larger than the buffer field, then the extra characters will be truncated (removed).

SECONDS()

calculates number of seconds elapsed since 00:00:00 GMT.

can be used as a random number seed.
*note: (not part of Standard Basic)
usage:

```
number% = SECONDS
```

SGN()

determines number's sign. If number is negative, returns -1. If number is positive, SGN returns 1. If number is zero, SGN returns 0.
usage:

```
sign = SGN(num)
```

example:

```
num = -99
sign = SGN(a)

IF sign = 0 THEN
    PRINT "sign equals zero"
ELSEIF sign < 0 THEN
    PRINT "sign is negative"
ELSEIF sign = 1 THEN
    PRINT "sign is positive"
ENDIF
```

SIN()

Returns the sine of a number. Number must be in radians.

```
float# = SIN(float#)
```

SINH(num)

calculates the hyperbolic sine of num
usage:

```
MyDouble# = SINH(aDouble#)
Print "SINH("; aDouble#; ")="; MyDouble#
```

SPACE$()

Returns a string containing number of blank spaces. Number must be in the range of 1 to 255.

```
A$ = SPACE$(20)
```

SQRT()

Returns the square root of a number. Number must be greater than zero.

```
float# = SQRT(float#)
```

STOP

Terminates program execution. Every program must have at lease one END/STOP/SYSTEM statement. The STOP command need not be the last line in the program. ie: it can be anywhere within the program.

```
           IF x = 0 THEN
                  GOTO TheEnd
           ENDIF
           ...
    TheEnd:
           STOP
```

END or STOP or SYSTEM can be used interchangeably.

STR$()

Converts a numeric value into an ascii character string. Number may be of any numeric data type.

```
string$ = STR$(99)
string$ = STR$(value!)
```

STRING$()

Returns a string containing the specified number of character. Number must be in the range of 1 to 255. Character may be any acsii code.

```
a$ = STRING$(10, "A")
a$ = STRING$(10, 65)
```

```
   a$ = STRING$(10, char%)
```

SWITCH

switch/case.

SYSTEM

Terminates program execution. Every program must have at lease one END/STOP/SYSTEM statement. The SYSTEM command need not be the last line in the program. ie: it can be anywhere within the program.

```
       IF x = 0 THEN
              GOTO TheEnd
       ENDIF
       ...
TheEnd:
       SYSTEM
```

SYSTEM or STOP or END can be used interchangeably.

TANH()

calculates the hyperbolic tangent of num.
usage:

```
MyDouble# = TANH(aDouble#)
Print "TANH("; aDouble#; ")="; MyDouble#
```

TIME$

retrieves the current time.
usage:

```
      mytime$ = TIME$
      PRINT mytime$
```

UCASE$()

Returns in the target string the upper case representation of the source string.

```
   ex:    string$ = UCASE$(string$)
          a$   = UCASE$(b$)
          b$ = "abc"
```

After executiong a$ = "ABC"

VAL()

Returns the numeric value of string.

```
   ex:    value = VAL(number$)
```

```
float# = VAL(number$)
```

WHILE/WEND

The WHILE/WEND is a loop construct with a logical expression at the start of the loop. The expression is examined at the startof each loop cycle.

```
ex:    WHILE x < 100
              x = abc + xyz
       WEND
```

WRITE#

Writes data to a sequential access file. Buffer is the file buffer number.

```
Ex:  WRITE #1, string$, integer%, float!, double#
```

Console

```
Usage examples:
===============

Here are some 'cut-n-paste' examples that you can try. These programs
can be found in the bxb_examples directory.
```

bx1.bas

```
Line Numbers, REM, CLS, PRINT, BEEP, END:
====================================
      In it's most basic from, the program statement looks like this:

      1 REM  bx1.bas version 1
      2 PRINT "hello world!"
      3 BEEP
      4 END


Line Numbers:
============
      Program lines may or may not be numbered. Line numbers have a
minimal     practical use and their usage is not recommended, but,
their usage is included f     or compatibility. If line numbers are
used, the must appear in column #1, the far l   eft  side,  only.  Line
numbers will appear here for illustrative purposes only.

      In the above example:
      REM:        in line 1: is a comment,
      PRINT:      in line 2: prints a message on the display screen
      BEEP:       in line 3: sounds a beep on the pc speaker
      END:        in line 4: terminates the program


      PRINT:      The PRINT command takes the form:
                   PRINT:     (generates a newline)
                  or      PRINT "text"
                  or      PRINT variable
                  or      PRINT (expression)

      END:     Every program must have an END command.
               However, it need not be at the end of the program.
```

bx2.bas

```
GOTO:
=====
      In the example below, using the GOTO command:
            line 4:          performs an absolute jump to line #8
            line 10:         an absolute jump to line #5
            line 7:          terminates the program
```

```
      1 REM  bx2.bas version 2
      2 PRINT "hello world!"
      3 BEEP
      4 GOTO 8
      5 PRINT "Back at line #6"
      6 PRINT "The End."
      7 END
      8 PRINT "Now at line #9"
      9 BEEP
      10 GOTO 5
```

bx3.bas

```
LET:
====
      The LET command has the form:
                         LET variable = value
             or      LET variable = variable
             or      LET variable = "text string"

      In this example,
            line 5:     keyword LET assigns a numeric value to integer
                              variable: "abc".
```

```
      1 REM  bx3.bas version 3
      2 CLS
      4 PRINT "hello world!"
      5 BEEP
      6 LET abc = 100
      7 PRINT "abc = 100"
      8 END
```

bx4.bas

Variable names are limited to 8 alpha-numeric characters only. No punctuation characters are permitted (at this time). Variables are distinguished into five types:

Symbol:	Type:
$	character string
#	double precision floating point
!	single precision float
%	long integer
none	integer (assumed)

If a variable name has no type specifier, it is assumed to be of type integer.

Here, in:

line 12 thru 14: three integer assignments are made.
line 15: variable 'abc' is re-assigned a new value:

```
1 REM  test.bas version 4
2 DIM abc = 0, xyz = 0, qwerty = 0
3 CLS
4 PRINT "hello world!"
5 BEEP
' ...
12 LET abc = 100
13 LET xyz = 999
14 LET qwerty = 12345
15 LET abc = 32123
16 END
```

bx5.bas

In this next example, variables: abc, xyz and qwerty are assigned values. Then PRINT command is used to display each variable's contents.

```
1 REM  test.bas version 5
2 CLS
3 PRINT "hello world!"
4 LET abc = 100
5 PRINT abc,
6 LET xyz = 999
7 LET qwerty = 12345
```

```
8 LET abc = 32123
9 PRINT xyz,
10 PRINT qwerty,
11 PRINT abc
12 PRINT "The End."
13 END
```

bx6.bas

```
LOCATE:
=======
The LOCATE command has the form:

        LOCATE row, column

In lines 4 and 5, variables: row and column are each given a
value.
Then on line 6, the LOCATE command it used to position the
cursor,
 at:
            Row: 2
         Column: 10

and again in line 10:


1  REM  test.bas version 6
2  CLS
3  PRINT "hello world!"
4  LET row = 2
5  LET column = 10
6  LOCATE row, column
7  PRINT "hello world!"
8  LET row = 4
9  LET column = 20
10 LOCATE row, column
11 PRINT "hello world!"
12 END

The console mode screen is character based and has the dimensions
of 25x80. The upper left corner is row 0, column 0.
```

bx7.bas

Math Expression Assignments:
============================
A variable assignment can be in the form of a mathematical expressions.
As shown in line 4:

```
1 REM  bx7.bas
2 CLS
3 PRINT "hello world!"
4 LET abc = 2 + 2
5 PRINT abc
6 END
```

bx8.bas

Complex algebraic expressions can be used in an assignment:

```
1  REM  test.bas version 8
2  CLS
3  PRINT "hello world!"
4  LET xylophone = 50
5  LET yazoo = 100
6  LET abc = yazoo/xylophone
7  LET xyz = yazoo/10
8  LOCATE abc , xyz
9  PRINT "hello world!"
10 LET quasar = 2
11 LET zapp = 4
12 LET abc = (quasar * quasar * zapp + zapp)/5
13 LET xyz = ((quasar*quasar)*zapp)+zapp
14 LOCATE abc, xyz
15 PRINT "hello world!"
50 END
```

bx9.bas

Optional: Line Numbers, REM, LET:
=================================
As stated previously, line numbers are entirely optional and in most cases un-needed. The only time they may actually be needed is as line labels. Here, even the REM is optional, as well. The REM keyword may be replaced with the apostrophy (') character. Blank or empty lines are ignored by the compiler and often times help to make the source code more readable. Also, the LET

keyword is optional.

```
'  bx9.bas
'
  CLS
  PRINT "hello world!"
  LET xylophone = 50
  LET yazoo = 100
  LET abc = yazoo/xylophone
  xyz = yazoo/10

  LOCATE abc , xyz
  PRINT "hello world!"
' ------------------------------------------
  quasar = 2
  zapp = 4
  abc = (quasar * quasar * zapp + zapp)/5
  xyz = ((quasar * quasar) * zapp) + zapp

  LOCATE abc, xyz
  PRINT "hello world!"
' ------------------------------------------

  PRINT:
  PRINT " 2*(3+4)*5/10 = ";
  abc = 2*(3+4)*5/10
  PRINT abc

TheEnd:
  END
```

In the above code, examine the PRINT statements.
PRINT has several forms:

```
  PRINT abc               prints variable followed by
newline
  PRINT:                   generates a newline
  PRINT ""                else a newline
  PRINT a;                semi-colon does not send a
newline
  PRINT a,                comma sends a tab, no newline
```

bx10.bas

BLOCK LABELS:
=============
"Block Labels" are preferred over line numbers and give symbolic meaning to a routine or block of code.

```
'  bx10.bas
  CLS
'               now jump to a block label
  GOTO OverThere
'
TheBeginning:
  PRINT "We're at The Beginning!"
  GOTO TheEnd
'
There:
  PRINT "We're There!"
  GOTO TheBeginning
'
JumpBack1:
  PRINT "We Jumped Back 1!"
  GOTO There
'
OverThere:
  PRINT "We're Over There!"
  GOTO JumpBack1
'
TheEnd:
  END
```

Where used, line Labels or "Block Labels" must start at the far left column, as shown above.

```
A Label name:
=============
*        may contain up to 32 alpha-numeric characters,
*        must contain no punctuation characters,
*        must be terminated by a colon (:),
*        is case sensitive, so:  label:

         Label:
and      LABEL:

are each unique.
```

When used with a GOTO statement, the label name is not terminated by the colon. i.e.:

```
        GOTO Label1
```

bx11.bas

```
'  test.bas version 11
Start1:
  CLS
  GOTO Jump
Return:
  GOTO TheEnd
Jump:
  PRINT "hello world!"
' ------------
  LET xylophone = 50
  LET yazoo = 100
  LET abc = yazoo/xylophone
  xyz = yazoo/10
  LOCATE abc , xyz
  PRINT "hello world!"
' ------------------------------------------
  quasar = 2
  zapp = 4
  abc = (quasar * quasar * zapp + zapp)/5
  xyz = ((quasar * quasar) * zapp) + zapp
  LOCATE abc, xyz
  PRINT "hello world!"
' ------------------------------------------
  PRINT:
  PRINT " 2*(3+4)*5/10 = ";
  abc = 2*(3+4)*5/10
  PRINT abc
  GOTO Return
TheEnd:
  END
```

bx12.bas

```
CLEAR:
======
The CLEAR command has the form:

        CLEAR
```

```
'  bx12.bas
'
Start1:
  CLS
  PRINT "hello world!"
' ------------
  LET xylophone# = 50.3
  LET yazoo# = 101.25
  LET abc = yazoo#/xylophone#
  xyz = yazoo#/10
  quasar = 2
  zapp = 4
  abc = (quasar * quasar * zapp + zapp)/5
  xyz = ((quasar * quasar) * zapp) + zapp
  abc = 2*(3+4)*5/10
'
  CLEAR
' ------------------------------------------
TheEnd:
  END
```

A CLEAR statement will erase all 'global' variables. If a variable no longer exists, or has previously been erased, the CLEAR command will ignore that fact. It will not cause an error. The CLEAR command has no effect on 'local' variables.

bx13.bas

```
'  bx13.bas
'
Start1:
  CLS
  PRINT "hello world!"
' ------------------------------------------
  abc = 11100
  xyz = 32000
  abcl% = 33000
  xyzl% = 99000
  abcf! = 33000.33
  xyzf! = 99000.47
  abcd# = 333000.33
  xyzd# = 999000.47
  test$ = "test"
'
  CLEAR
' ------------------------------------------
TheEnd:
  END
```

bx14.bas

```
NUMERICAL DATA TYPES:
=====================
```

**Note: all variable names must be unique, regardless of type.

**Hungarian notation: the term refers to prefixing a variable name with a letter signifier as to the variables type. ie: intiger, long, float, double, string. In cases where two or more variables of different types, share the same name, Hungarian notation can be used.

**reverse-Hungarian notation: where the data type signifier is added to the end of the variable name.

Simple integers have a limited range and can not handle fractional or "real" numbers. As shown here, an expression may contain mixed data types. The result will be the destination variable's type.

```
'  bx14.bas
'
  DIM xylophone# = 0, yazoo# = 0, abc = 0
  DIM xyz = 0, quasar = 0, zapp = 0
'
Start1:
  CLS
  PRINT "hello world!"
' ------------
  LET xylophone# = 50.3
  LET yazoo# = 101.25
  LET abc = yazoo#/xylophone#
  xyz = yazoo#/10
  LOCATE abc , xyz
  PRINT "hello world!"
' ------------------------------------------
  quasar = 2
  zapp = 4
  abc = (quasar * quasar * zapp + zapp)/5
  xyz = ((quasar * quasar) * zapp) + zapp
  LOCATE abc, xyz
  PRINT "hello world!"
' ------------------------------------------
  PRINT:
  PRINT " 2*(3+4)*5/10 = ";
  abc = 2*(3+4)*5/10
  PRINT abc
' ------------------------------------------
  PRINT:
  PRINT "xylophone# = ";
  PRINT xylophone#
  PRINT "yazoo# = ";
  PRINT yazoo#
TheEnd:
  CLEAR
  END
```

bx15.bas

Here are some examples of expressions using different data types.

```
'  bx15.bas
'
'
Start1:
  CLS
  PRINT "hello world!"
' ------------------------------------------double float
  LET xylophone# = 50.3
  LET yazoo# = 101.25
  LET abc = yazoo# / xylophone#
  xyz = yazoo# / 10
  LOCATE abc , xyz
  PRINT "hello world!"
' ------------------------------------------long integers
  quasar% = 2
  zapp% = 4
  abcl% = (quasar% * quasar% * zapp% + zapp%)/5
  xyzl% = ((quasar% * quasar%) * zapp%) + zapp%
  LOCATE abcl%, xyzl%
  PRINT "hello world!"
' ------------------------------------------
  PRINT:
  PRINT " 2*(3+4)*5/10 = ";
  abc = 2*(3+4)*5/10
  PRINT abc
' ------------------------------------------
  PRINT:
  PRINT "xylophone# = ";
  PRINT xylophone#
  PRINT "yazoo# = ";
  PRINT yazoo#
  PRINT:
  PRINT "abcl%=";
  PRINT abcl%
  PRINT "xyzl%=";
  PRINT xyzl%
' ------------------------------------------
TheEnd:
  CLEAR
  END
```

bx16.bas

```
'  bx16.bas
'
Start1:
  CLS
' --------------------------------------------
  abc = 11100
  xyz = 32000
  abcl% = 33000
  xyzl% = 99000
  abcf! = 33000.33
  xyzf! = 99000.47
  abcd# = 333000.33
  xyzd# = 999000.47
'
'                  integer
  PRINT:
  PRINT "abc=";
  PRINT abc
  PRINT "xyz=";
  PRINT xyz
'                  long
  PRINT "abcl%=";
  PRINT abcl%
  PRINT "xyzl%=";
  PRINT xyzl%
'                  float
  PRINT "abcf!=";
  PRINT abcf!
  PRINT "xyzf!=";
  PRINT xyzf!
'                  double
  PRINT "abcd#=";
  PRINT abcd#
  PRINT "xyzd#=";
  PRINT xyzd#
' --------------------------------------------
TheEnd:
  CLEAR
  END
```

bx17.bas

```
'  bx17.bas
'
  xylophone# = 0
  yazoo# = 0
  abc = 0
  xyz = 0
  quasar% = 0
  zapp% = 0
  abcl% = 0
  xyzl% = 0
  abcf! = 0
  xyzf! = 0
  abcd# = 0
  xyzd# = 0
'
Start1:
  CLS
  PRINT "hello world!"
' ------------------------------------------double float
  LET xylophone# = 50.3
  LET yazoo# = 101.25
  LET abc = yazoo# / xylophone#
  xyz = yazoo# / 10
  LOCATE abc , xyz
  PRINT "hello world!"
' ------------------------------------------long integers
  quasar% = 2
  zapp% = 4
  abcl% = (quasar% * quasar% * zapp% + zapp%)/5
  xyzl% = ((quasar% * quasar%) * zapp%) + zapp%
  LOCATE abcl%, xyzl%
  PRINT "hello world!"
' ------------------------------------------
  PRINT:
  PRINT " 2*(3+4)*5/10 = ";
  abc = 2*(3+4)*5/10
  PRINT abc
' ------------------------------------------
  abc = 11100
  xyz = 32000
  abcl% = 33000
  xyzl% = 99000
  abcf! = 33000.33
  xyzf! = 99000.47
  abcd# = 333000.33
  xyzd# = 999000.47
'                  integers
  PRINT:
  PRINT "abc=";
  PRINT abc
  PRINT "xyz=";
  PRINT xyz
```

```
'                   long integers
  PRINT "abcl%=";
  PRINT abcl%
  PRINT "xyzl%=";
  PRINT xyzl%
'                   float
  PRINT "abcf!=";
  PRINT abcf!
  PRINT "xyzf!=";
  PRINT xyzf!
'                   double
  PRINT "abcd#=";
  PRINT abcd#
  PRINT "xyzd#=";
  PRINT xyzd#
' ------------------------------------------
TheEnd:
  CLEAR
  END
```

bx18.bas

STRING VARIABLES:
=================
String variable assignments may be expressed in several ways. To begin with, a string variable is terminated by the '$' symbol. A string variable may be a single character, no character, or a number of characters, with a maximum of 255 characters.

```
'  bx18.bas
'
  abc$ = ""
  xyz$ = ""
'
Start1:
  CLS
  PRINT "hello world!"
  abc$ = "test"
  xyz$ = ""
'   ^---------------here, xyz is null, empty!
  END
```

bx19.bas

```
'  bx19.bas
'
  abc$ = ""
  xyz$ = ""

  CLS
  abc$ = "testing"
  xyz$ = abc$
  PRINT abc$
  PRINT xyz$
' --------------------------------------------
TheEnd:
  END
```

bx20.bas

```
COMPLEX PRINTING:
=================
The PRINT statement can be used in complex ways to create the
desired display.
```

```
'  bx20.bas
'
  abc = 0
  xyz = 0
  abcl% = 0
  xyzl% = 0
  abcf! = 0
  xyzf! = 0
  abcd# = 0
 xyzd# = 0
'
Start1:
  CLS
  PRINT "hello world!"
' --------------------------------------------
  abc = 2*(3+4)*5/10
  PRINT "": " 2*(3+4)*5/10 ="; abc
' --------------------------------------------
  abc = 11100
  xyz = 32000
  abcl% = 33000
  xyzl% = 99000
  abcf! = 33000.33
  xyzf! = 99000.47
  abcd# = 333000.33
  xyzd# = 999000.47
'                                               integers
```

```
  PRINT "": "abc="; abc: "xyz="; xyz
'                                                long integers
  PRINT "abcl%="; abcl%: "xyzl%="; xyzl%
'                                                float
  PRINT "abcf!="; abcf!: "xyzf!="; xyzf!
'                                                double
  PRINT "abcd#="; abcd#: "xyzd#="; xyzd#
' -----------------------------------------
TheEnd:
  CLEAR
  END
```

bx21.bas

```
GOSUB RETURN:
=============
Subroutines are the preferred method of program branching.
Subroutines allow for a more structured style of programming over
the absolute jump of a GOTO statement. Subroutines are identified
by a block label and called with the GOSUB command. Each GOSUB
command has a matching RETURN command. If you GOSUB, you must
RETURN from the subroutine.
```

```
'  bx21.bas
'
  hello$ = ""
  test$ = ""
  abc = 0
  xyz = 0
  yazoo# = 0
  xylophone# = 0
  quasar% = 0
  zapp% = 0
  abcl% = 0
  xyzl% = 0
  abcf! = 0
  xyzf! = 0
  abcd# = 0
  xyzd# = 0
'
  GOSUB Start1
  GOSUB DoubleFloat
  GOSUB LongIntegers
  GOSUB RDParser
  GOSUB PrintVars
  GOSUB ClearVars
  GOTO TheEnd
```

```
' ------------------------------------------
Start1:
  CLS
  hello$ = "hello world!"
  PRINT hello$
  RETURN
' ------------------------------------------double float
DoubleFloat:
  LET xylophone# = 50.3
  LET yazoo# = 101.25
  LET abc = yazoo# / xylophone#
  xyz = yazoo# / 10
  LOCATE abc , xyz
  PRINT hello$
  RETURN
' ------------------------------------------long integers
LongIntegers:
  quasar% = 2
  zapp% = 4
  abcl% = (quasar% * quasar% * zapp% + zapp%)/5
  xyzl% = ((quasar% * quasar%) * zapp%) + zapp%
  LOCATE abcl%, xyzl%
  PRINT hello$
  RETURN
' ------------------------------------------
RDParser:
  abc = 2*(3+4)*5/10
  PRINT "": " 2*(3+4)*5/10 ="; abc
  RETURN
' ------------------------------------------
PrintVars:
  abc = 11100
  xyz = 32000
  abcl% = 33000
  xyzl% = 99000
  abcf! = 33000.33
  xyzf! = 99000.47
  abcd# = 333000.33
  xyzd# = 999000.47
'                                         integers
  PRINT "": "abc="; abc: "xyz="; xyz
'                                         long integers
  PRINT "abcl%="; abcl%: "xyzl%="; xyzl%
'                                         float
  PRINT "abcf!="; abcf!: "xyzf!="; xyzf!
'                                         double
  PRINT "abcd#="; abcd#: "xyzd#="; xyzd#
  RETURN
' ------------------------------------------
ClearVars:
  test$ = "test"
  CLEAR
  RETURN
' --------------------------------------------
TheEnd:
  END
```

bx22.bas

GOSUB's may be nested. That means that a second GOSUB may be called even before RETURNing from the first subroutine. The second subroutine may even call a third subroutine, etc. As long as there are an equal number of RETURNs as GOSUBs.

```
'  bx22.bas
'
  hello$ = ""
  test$ = ""
  xylophone# = 0
  yazoo# = 0
  abc = 0
  xyz = 0
  quasar% = 0
  zapp% = 0
  abcl% = 0
  xyzl% = 0
  abcf! = 0
  xyzf! = 0
  abcd# = 0
  xyzd# = 0
'
'
  GOSUB Start1
  GOTO TheEnd
' --------------------------------------------
Start1:
  CLS
  hello$ = "hello world!"
  PRINT hello$
  GOSUB DoubleFloat
  RETURN
' --------------------------------------------double float
DoubleFloat:
  LET xylophone# = 50.3
  LET yazoo# = 101.25
  LET abc = yazoo# / xylophone#
  xyz = yazoo# / 10
  LOCATE abc , xyz
  PRINT hello$
  GOSUB LongIntegers
  RETURN
' ------------------------------------------long integers
LongIntegers:
  quasar% = 2
  zapp% = 4
  abcl% = (quasar% * quasar% * zapp% + zapp%)/5
  xyzl% = ((quasar% * quasar%) * zapp%) + zapp%
  LOCATE abcl%, xyzl%
  PRINT hello$
  GOSUB RDParser
  RETURN
' ------------------------------------------
RDParser:
  abc = 2*(3+4)*5/10
  PRINT "": " 2*(3+4)*5/10 ="; abc
  GOSUB PrintVars
  RETURN
' ------------------------------------------
PrintVars:
  abc = 11100
  xyz = 32000
  abcl% = 33000
  xyzl% = 99000
  abcf! = 33000.33
  xyzf! = 99000.47
  abcd# = 333000.33
  xyzd# = 999000.47
'                               integers
  PRINT "": "abc="; abc: "xyz="; xyz
'                               long integers
  PRINT "abcl%="; abcl%: "xyzl%="; xyzl%
'                               float
  PRINT "abcf!="; abcf!: "xyzf!="; xyzf!
'                               double
  PRINT "abcd#="; abcd#: "xyzd#="; xyzd#
  GOSUB ClearVars
  RETURN
' ------------------------------------------
ClearVars:
  test$ = "test"
  CLEAR
  RETURN
' ------------------------------------------
TheEnd:
  END
```

bx23.bas - FOR/NEXT

```
FOR NEXT:
```

=========

The FOR/NEXT commands allow for the creation of complex loop structures.

The basic syntax is:

```
        FOR (condition, increment)
                program...
        NEXT (increment loop)
```

The 'conditional expression' and 'increment' take the form of:

```
        variable% = start TO end, STEP increment

      variable:  is the object of the condition
         start:  is the starting value for 'variable'
           end:  is the destination for 'variable'
     increment:  is the amount by which to increment 'variable'
```

```
Where:     FOR x = 1 TO 100 STEP 1

    x = 1:   Assigns the starting value of 1 to variable 'x'.
             Variable 'x' must be of type: INTEGER.
   TO 100:   Sets a desired value of: 100 for 'x'.
   STEP 1:   The amount by which to increment.
             If the STEP value were: 3
             'x' would be incremented by 3 after each loop cycle.
             If no STEP value is stated, the default is: 1.
```

NEXT is little more than a block label, or block delimiter, but, it also keeps track of which variable we need to increment for the next cycle.

FOR/NEXTs may be nested.
i.e.:

```
        FOR (cond-1)             [outer loop]
            FOR (cond-2)         [inner loop]
                program...
            NEXT (inc)           [inner loop]
        NEXT (inc)               [outer loop]
```

However, you may not exit an outer loop before exiting an inner loop. An inner loop may be terminated, however. To terminate an inner loop, an IF/ELSE expression can be used to test for a condition and based on the result, object variable can be forced to the destination value.

Example:

```
        FOR x = 1 TO 10
            FOR y = 1 TO 100    ......[inner loop]
                IF y = 50 THEN
```

```
                    y = 100           [force: y = 100]
                ENDIF
                PRINT CHR$(32);
            NEXT y              ......[inner loop]
        NEXT x
```

This will have the desired effect of satisfying the condition:

```
        FOR y = 1 TO 100
```

```
'  bx23.bas
'
  x = 0
  y = 0
'
  CLS
  GOSUB TOP
  GOSUB Center
  GOSUB Bottom
  GOTO TheEnd
'-----------------------
Center:
  FOR x = 1 TO 5
      PRINT "*";
      FOR y = 1 TO 28
          PRINT " ";
      NEXT y
      PRINT "*":
  NEXT x
  RETURN
'-----------------------
TOP:
Bottom:
  FOR x = 1 TO 30
      PRINT "*";
  NEXT x
  PRINT "":
  RETURN
'-----------------------
TheEnd:
  END
```

This example will generate a box made of stars in the upper left corner of the display, like this:

```
******************************
*                            *
*                            *
*                            *
*                            *
*                            *
```

```
******************************
```

bx24.bas - POWER/ MODULUS

```
POWER, MODULUS:
===============
Besides the numerical operators: +,-,/ and *, Bxbasic also uses
the Power symbol: ^ and the Modulo symbol: %.

Power:
======
The symbol ^ represents the Power function. The effect is to
raise a number to the Power of a second number.

Example:
        result = 10 ^ 2    "result" equals: 10 raised to the
                           power of 2.

                           The product, (100) is assigned to
                           "result"


Modulo:
=======
The symbol % represents the Modulus function. The effect is to
capture the remainder of a division operation.

Example:
        mod = 10 % 3  "mod" equals the remainder of 10/3.

                      The product, (1) is assigned to variable
                      "mod"
```

```
'  bx24.bas`
'
  power = 0
  ten = 0
  three = 0
  mod = 0
'
  GOSUB Start1
  END
' ------------------------------------------
Start1:
  CLS
  power = 10 ^ 2
  PRINT "power="; power
'
  ten = 10
  three = 3
'                we can use the modulus operator: %
  mod = ten % three
  PRINT "mod="; mod
'                or we can use the keyword: MOD
  mod = ten MOD (30 MOD 9)
  PRINT "mod="; mod
  RETURN
```

The word MOD may be used in place of the % symbol. One word of caution: when used with numeric variables, the % symbol has to have a blank space on either side of it, or the variable may be confused as a LONG integer.

```
i.e.:    var1 % var2     correct.
         var1%var2      incorrect!
```

Bx25.bas - IF/ELSE

```
IF/ELSE:
========
The IF/ELSE conditional expression uses the form:
        IF (condition) THEN             first condition
                [program...]
        ELSEIF (condition) THEN         second condition
                [program...]
        ELSE                            default action
                [program...]
        ENDIF                           terminator
```

In the above, there are two conditional expressions:

```
Condition #1:
        if condition #1 is TRUE, then:
                [the following action is taken]
                [program resumes at point beyond the ENDIF]
        if condition #1 is FALSE, then:
                [evaluate expression #2]

Condition #2:
        elseif condition #2 is TRUE, then:
                [the following action is taken]
                [program resumes at point beyond the ENDIF]
        elseif condition #2 is FALSE, then:
                [take default action]
```

```
Default:
        else
            [the following action is taken]
            [program resumes at point beyond the ENDIF]
        endif
```

The simplest conditional expression might be:

```
        IF (condition) THEN        conditional expression
            [program...]
        ENDIF               terminator
```

where only one condition is evaluated.

Conditional expressions may also use the Boolean AND/OR operators.

```
Example:
        IF a = b AND x = y THEN
            ...etc.
or:
        IF a = b OR x = y THEN
            ...etc.
```

These condition operators may be used:

operator	description
=	is equal to
<>	is not equal to
<	is less than
>	is greater than
<=	is less or equal to
>=	is greater or equal to

```
'  bx25.bas
'
  CLS
  abc = 99
  xyz = 33
  abcs$ = "test"
  xyzs$ = "testing"
'
'
  IF xyzs$ = "testing" AND abc >= xyz THEN
     PRINT "if:expression = true"
  ELSEIF abc <= 100 OR abcs$ <= "hello" THEN
     PRINT "elseif:expression = true"
  ELSE
     PRINT "else:expressions = false"
  ENDIF
  PRINT "done"
```

```
' ------------------------------------------
TheEnd:
  END
```

bx26.bas - STRING FUNCTIONS

```
STRING FUNCTIONS:
=================
       Function:  Usage:
       =========  ============================================
        CHR$()    a$ = CHR$(n)
                           where: (n) is an ascii value.

        LEFT$()   a$ = LEFT$(s$,n)
                           where: s$ is a string variable name,
                           'n' is the number of characters to
                           copy from the left.

        RIGHT$()  a$ = RIGHT$(s$,n)
                           where: s$ is a string variable name,
                           'n' is the number of characters to
                           copy from the right.

        MID$()    a$ = MID$(s$,x,n)
                           where: s$ is a string variable name,
                           'x' is the character starting position
                           from the left,
                           'n' is the number of characters to
                           copy.

        SPACE$()  a$ = SPACE$(n)
                           where: 'n' is the number of blank
                           spaces

        STR$()    a$ = STR$(n)
                           where: 'n' is a numeric value to be
                           converted to a character string.

        STRING$() a$ = STRING$(n,x)
                           where: 'n' is the number of
                           characters,
                           'x' is an ascii value.

        UCASE$()  a$ = UCASE$(s$)
                           where: s$ is a string variable name.

        LCASE$()  a$ = LCASE$(s$)
                           where: s$ is a string variable name.
```

```
'  bx26.bas
'
  xyzi = 0
  abcs$ = ""
  xyzs$ = ""
'
'
  CLS
  xyzi = 42
  abcs$ = CHR$(xyzi)
  PRINT "abcs$ = ";abcs$
'
  xyzs$ = "testing"
  abcs$ = LEFT$(xyzs$, 4)
  PRINT abcs$
'
  abcs$ = RIGHT$(xyzs$, 5)
  PRINT abcs$
'
  abcs$ = MID$(xyzs$, 3, 3)
  PRINT abcs$
'
  abcs$ = SPACE$(3)
  PRINT ">";abcs$;"<"
'
  abcs$ = STR$(199)
  PRINT abcs$
'
  abcs$ = STRING$(10, xyzi)
  PRINT ">";abcs$;"<"
' ----------------------------------------
TheEnd:
  END
```

bx27.bas - STRING FUNCTIONS

```
'  bx27.bas
'
  ixyz = 0
  sxyz$ = ""
  sabc$ = ""
'
'
  CLS
  ixyz = 42
  sxyz$ = "testing"
'
```

```
  sabc$ = LEFT$(sxyz$, 4) + CHR$(ixyz) + RIGHT$(sxyz$, 5) +
SPACE$(3)
  sabc$ = sabc$ + MID$(sxyz$, 3, 3) + STR$(-199) + STRING$(10,
ixyz)

  PRINT sabc$
'
  sabc$ = CHR$(34) + "Hello" + CHR$(32) + "world!" + CHR$(34)
  PRINT sabc$
' ------------------------------------------
TheEnd:
  END
```

bx28.bas - STRING FUNCTIONS

```
'  bx28.bas
'
  abc$ = ""
'
  CLS
  abc$ = "This is a test of the Emergency Broadcast System"
  PRINT "Test: >"; CHR$(251); "< End Test"
  PRINT ">"; LEFT$(abc$, 14); "<"
  PRINT ">"; RIGHT$(abc$, 26); "<"
  PRINT ">"; MID$(abc$, 23, 19); "<"
  PRINT ">"; SPACE$(10); "<"
  PRINT ">"; STR$(1000); "<"
  PRINT ">"; STRING$(10, 251); "<"
' ------------------------------------------
TheEnd:
  END
```

bx29.bas - STRING FUNCTIONS

```
'  bx29.bas
'
  abc = 0
  xyz = 0
'
  CLS
  PRINT ">"; STR$(1000); "<"
  PRINT ">"; STRING$(10, 251); "<"
  PRINT CHR$(247)
  abc = 10
  xyz = 3
  PRINT abc * xyz
  PRINT 1 + (abc * xyz)
  PRINT (abc / xyz) * 2
  PRINT 1+(2*5)/3
' -------------------------------------------
TheEnd:
  END
```

bx30.bas - INKEY$

```
INKEY$:
=======
```

The INKEY$ function captures the character currently entered in the keyboard buffer. INKEY$ has a unique quality that makes it quite different from other types of keyboard input. When INKEY$ is called, it does not wait for the user to strike a key. Instead, after collecting the character, even if there is no character in the buffer, it continues on to the next instruction.

INKEY$ is especially well suited for a loop construct or a subroutine call that polls the keyboard buffer.

```
'  bx30.bas
'
  abc$ = ""
'
  CLS
Start:
  abc$ = INKEY$
  IF abc$ = "" THEN
    GOTO Start
  ENDIF
  PRINT abc$
' -------------------------------------------
TheEnd:
  END
```

In the above example, INKEY$ is used in a continuous loop, that tests the keyboard buffer for a character. If there is no character, then abc$ will be null, or empty. In that event, program control jumps back up to START: and repeats the loop.

bx31.bas - INKEY$

```
'  bx31.bas
'
  abc$ = ""
'
  CLS
  PRINT "Press any key: ";
Start:
  abc$ = INKEY$
  IF abc$ = "" THEN
    GOTO Start
  ENDIF
  PRINT abc$
' ------------------------------------------
TheEnd:
  END
```

bx32.bas - INPUT

```
INPUT:
======
The INPUT command accepts standard input from the keyboard. The
INPUT command requires a return or newline character to terminate
input.

Example:
        INPUT a$        accepts a character string
        INPUT val       accepts a numeric value

In this example:
        INPUT ;"First: "; first$; " Last: "; last$:

the display shows:
        First: _

while the cursor waits for the contents of first$ to be entered.
Which then displays:
        First: Bill Last:_

on the same line, waiting for the contents of last$ to be
entered. INPUT, in this example, acts as both an INPUT and a
PRINT command. Since the return key is required to end input, by
```

placing a semi-colon after the INPUT command, tells Bxbasic not to echo the newline character.

Example:
```
        INPUT ; "Enter your Name: "; name$
```

This will have the effect of keeping the cursor on the same line.

```
'  bx32.bas
'
  abc$ = ""
  first$ = ""
  init$ = ""
  last$ = ""
  age = 0
  mo = 0
  day = 0
  year% = 0
'
'
  CLS
  PRINT "Press any key to begin: ";
Start:
  abc$ = INKEY$
  IF abc$ = "" THEN
    GOTO Start
  ENDIF
  PRINT abc$
'
  PRINT "Enter your name:"
  INPUT ;"First: "; first$; " Initial: "; init$; " Last: ";
last$:
  PRINT "Enter your age and birth date:"
  INPUT ;"Age: "; age; " Month: "; mo; "/Day: "; day; "/Year: ";
year%:
'
  PRINT first$, init$, last$
  PRINT age, mo, day, year%
' ------------------------------------------
TheEnd:
  END
```

bx33.bas - INPUT$

INPUT$:

=======
The INPUT$ function differs from the INPUT command in the way in which it is used. The INPUT$ function is a character string function and is used to input string data, as opposed to numeric data. The characters entered are assigned to a string variable. The INPUT$ function accepts a parameter, in the form of an integer value. That value is the number of characters that
INPUT$ will accept.

Example:

```
        a$ = INPUT$(n)     where 'n' is an integer
```

Once the total number of characters have been entered, the program moves on to the next instruction. Hitting the return key is not required, but, entering the total number of characters is. An example of where this might be used is in accepting data entry of a known fixed length.

```
'  bx33.bas
'
  abc$ = ""
'
  CLS
  PRINT "Enter 10 digits"
  abc$ = INPUT$(10)
  PRINT "": abc$
' -------------------------------------------
TheEnd:
  END
```

bx34.bas - LINE INPUT

LINE INPUT:
===========
The LINE INPUT command is very similar to the plain INPUT command, except that all the data entered is assigned to a single string variable. Also, LINE INPUT will accept an entire line of text, up to 255 characters and is terminated by a newline character. LINE INPUT can be used much the same way as the INPUT command, with or without a prompt string:

Example:

```
        LINE INPUT "Enter your: Address, City and State: "; a$
```

```
or      LINE INPUT a$
```

The return key terminates input.

```
'  bx34.bas
'
  abc$ = ""
'
  CLS
  PRINT "1: ";
  LINE INPUT abc$
'           ^-----notice no prompt
  PRINT abc$
  LINE INPUT "2:enter a string: "; abc$
  PRINT abc$
  LINE INPUT ;"3:enter a string: "; abc$
'            ^------no echo:return
  PRINT abc$
  LINE INPUT ;"4:enter a string: "; abc$,
'  no echo---^         insert tab-------^
  PRINT abc$
' ------------------------------------------
TheEnd:
  END
```

bx35.bas - ALGEBRAIC FUNCTIONS

```
ALGEBRAIC FUNCTIONS:
====================
        Function:     Usage:
       ==========  ============================================
        ABS(n)      num% = ABS(number)
                        returns the absolute value of a number.

        ASC(c$)  num  = ASC(char$)
                        returns the value of an ascii character.

        ATN(n)   num# = ATN(num)
                        returns the arctangent of a number.

        COS(n)   num# = COS(num)
                        returns the cosine of a number.

        SIN(n)   num# = SIN(num)
```

```
                   returns the sine of a number.

        TAN(n)   num# = TAN(num)
                   returns the tangent of a number.

        SQRT(n)  num# = SQRT(num)
                   returns the square root of a number.

        INT(n)   num% = INT(num#)
                   returns the integer value of a floating
                   point number.

        VAL(s$) num  = VAL(str$)
                num  = VAL("100")
                   returns the value of a numeric string
                   in a variable or quoted string.

        LOG(n)  num  = LOG(n)
                   returns the natural logarithm of number.

        LOG10(n) num = LOG10(n)
                   returns the logarithm base 10 of number.
```

```
'  bx35.bas
'
  ixyz = 0
  iabc = 0
  labc! = 0
'
  CLS
  ixyz = 99
  iabc = ABS(ixyz - 1.75)
  PRINT iabc
  iabc = ASC("A")
  PRINT iabc
  labc! = ATN(ixyz / 3)
  PRINT labc!
  labc! = COS(5.8 * .0174533)
  PRINT labc!
  labc! = SIN(ixyz / 11)
  PRINT labc!
  labc! = TAN(ixyz / 10)
  PRINT labc!
  labc! = SQRT(ixyz)
  PRINT labc!
  iabc = INT(ixyz / 3.1)
  PRINT iabc
```

```
' ------------------------------------------
TheEnd:
  END
```

bx36.bas - ALGEBRAIC FUNCTIONS

```
'  bx36.bas
'
  abc = 10
  xyz = 99
'
  CLS
'
  PRINT ABS(xyz - 1.75)
  PRINT ASC("A")
  PRINT ATN(xyz / 3)
  PRINT COS(5.8 * .0174533)
  PRINT SIN(xyz / 11)
  PRINT TAN(xyz / 10)
  PRINT SQRT(xyz)
  PRINT INT(xyz / 3.1)
' ------------------------------------------
TheEnd:
  END
```

bx37.bas - DISK I/O

```
DISK FILE I/O:
==============
OPEN, CLOSE:
============
File I/O is dependant on the abiliy to OPEN and CLOSE disk files. The
OPEN command requires a specific format using three parameters.

i.e.:
        1) I/O Mode
        2) File Handle
        3) Filename/Path

Example:
            OPEN "I", #1, "data.fil"
    I/O Mode------^    ^      ^-------Filename
```

```
                       ^-----File Handle
```

The I/O Modes are:

```
        I = Input        read a file
        O = Output       create/write to file
        A = Append       write to existing file
```

File Handle numbers can range from 1 to 99.
The CLOSE command can be used in two ways:

```
        CLOSE                close all open files
        CLOSE 1,2,3          close files #1, #2 and #3
```

Before we can run this next program, you will need to create a file named Test.txt. Using Notepad, create an empty file, just hit the return key about three or four times, then save the file to your working directory, naming it "Test.txt", and close it.

```
'  bx37.bas
  CLS
'
  PRINT "Opening Test File"
  OPEN "I", #1, "test.txt"
  CLOSE 1
' ------------------------------------------
TheEnd:
  END
```

Okay, maybe that wasn't all that impressive, but, it did OPEN Test.txt for Input and then CLOSE it.

In this next example, a character string, filename$, is assigned the file-name information. Either a quoted string, as used in the above, or a string variable may be used for this purpose.

bx38.bas - OPEN, CLOSE

```
'  bx38.bas
'
  DIM filename$ = "test.txt"
'
  CLS
```

```
'
  PRINT "Opening Test File"
  OPEN "I", #1, filename$
  CLOSE 1
' ------------------------------------------
TheEnd:
  END
```

bx39.bas - OPEN, INPUT, CLOSE

For the next example, copy this line of data to our data file, Test.txt:

```
"hello world","next string",32000,650000,1.123,3000000.123
```

Make sure this is at the very top line. Also, delete any blank lines that may come after it, so that this is the only thing in Test.txt.

```
'  bx39.bas
'
  input$ = ""
  next$ = ""
  valuea = 0
  valueb% = 0
  valuec! = 0
  valued# = 0
'
  CLS
'
  PRINT "Opening Test File"
  OPEN "I", #1, "test.txt"
  INPUT#1, input$, next$, valuea, valueb%, valuec!, valued#
  CLOSE 1
  PRINT input$, next$, valuea, valueb%, valuec!, valued#
' ------------------------------------------
TheEnd:
  END
```

bx40.bas - OPEN, INPUT, CLOSE

When executed, Bxbasic OPENs Test.txt for Input, using Handle #1 and INPUTs two string variables and four numeric variables. It then CLOSEs Handle #1 and prints out the information.

Now, let's try something a little more challenging. Copy the following into the data file: Test.txt

```
"hello world","next string",32000,650000,1.123,3000000.123
"hello ","world",2000,50000,0.123,5000000.123
```

```
'  bx40.bas
'
  input$ = ""
  next$ = ""
  valuea = 0
  valueb% = 0
  valuec! = 0
  valued# = 0
'
  CLS
'
  PRINT "Opening Test File"
  OPEN "I", #1, "test.txt"
  INPUT#1, input$, next$, valuea, valueb%, valuec!, valued#
  PRINT input$, next$, valuea, valueb%, valuec!, valued#
'
  INPUT#1, input$, next$, valuea, valueb%, valuec!, valued#
  PRINT input$, next$, valuea, valueb%, valuec!, valued#
  CLOSE 1
' ------------------------------------------
TheEnd:
  END
```

In this example, Bxbasic read-in two seperate lines of data and processed it as before. This is okay, if you only have one or two lines of data to input. In practice though, it's not very practical.

bx41.bas - EOF()

EOF():
======
If you have a data file that contains twenty lines of data to input, you don't want to code twenty identical lines of INPUT

instructions. Instead, a more practical application would be to read the data in, in some form of loop.

For example:

```
  OPEN "I", #1, "test.txt"
'
Start:
  INPUT#1, input$, next$, valuea, valueb%, valuec!, valued#
  PRINT input$, next$, valuea, valueb%, valuec!, valued#
  GOTO Start
  ...
```

Here, the program loops through the data file, inputing each line of data. There is one major problem with this though. The code above forms a continuous loop that would end in an error or program crash.

It has to do with the fact that when the program gets to the end of the file, it just tries to keep on reading. This is not a good thing. The solution is to have a means of detecting when we have reached the end of the file and terminate the process before it does any harm. That is where EOF comes in.

EOF is a file or device function. The sole purpose is to detect the End-Of-File. Using the sample code from above, all we need to do is to add a condition test to the loop, that will test for End-Of-File.

Like this:

```
  OPEN "I", #1, "test.txt"
'
Start:
  IF EOF(1) THEN
     GOTO Finish
  ENDIF
  INPUT#1, input$, next$, valuea, valueb%, valuec!, valued#
  PRINT input$, next$, valuea, valueb%, valuec!, valued#
  GOTO Start
Finish:
  CLOSE 1
```

In this example, we keep looping, reading-in data, until we detect an End-Of-File condition. Then we stop trying to read.

Here is a Test.bas that we can try it on. Before we can test this out, we need to modify Test.txt. Copy the following into Test.txt:

```
"hello world","next string", 32000, 650000, 1.123, 3000000.123
"hello ","world", 2000, 50000, 0.123, 5000000.123
"hello world","next string", 32000, 650000, 1.123, 3000000.123
```

Save and close it. Now execute it.

```
'  bx41.bas
'
  input$ = ""
  next$ = ""
  valuea = 0
  valueb% = 0
  valuec! = 0
  valued# = 0
'
  CLS
  PRINT "Opening Test File"
  OPEN "I", #1, "test.txt"
'
Start:
  IF EOF(1) THEN
     GOTO Finish
  ENDIF
  INPUT#1, input$, next$, valuea, valueb%, valuec!, valued#
'
  IF input$="" THEN
     GOTO Start
  ENDIF
  PRINT input$, next$, valuea, valueb%, valuec!, valued#
  GOTO Start
'
Finish:
  CLOSE 1
' ------------------------------------------
TheEnd:
  END
```

In line 11, is another status test. There, it is testing for the possibility of a mis-read, where a read attempt was made but for some reason it was unable to. In this case, it loops back up to see if it has reached the End-Of-File.

Bx42.bas - Write Disk I/O

WRITE:
======
Inputing data is only half of file I/O. The other half is WRITEing data to a disk file. The WRITE command looks identical to the INPUT command. In fact, the WRITE command is the mirror image of the INPUT command. Except that the data flows in the opposite direction.

Here is an example where we open a second file for OUTPUT and WRITE to it the data we have INPUT from the first file. Notice that the data written out need not be in the same order as was read-in.

```
'  bx42.bas
'
  nput$ = ""
  next$ = ""
  valuea = 0
  valueb% = 0
  valuec! = 0
  valued# = 0
'
  CLS
'
  PRINT "Opening Test File"
  OPEN "I", #1, "test.txt"
'
  OPEN "O", #2, "test2.txt"
'
Start:
  IF EOF(1) THEN
     GOTO Finish
  ENDIF
  INPUT#1, input$, next$, valuea, valueb%, valuec!, valued#
  IF input$="" THEN
     GOTO Start
  ENDIF
  PRINT input$, next$, valuea, valueb%, valuec!, valued#
'
  WRITE#2, valued#, valuec!, valueb%, valuea, next$, input$
' ..................write data in reversed order!
  GOTO Start
'
Finish:
  CLOSE 1, 2
```

```
' -------------------------------------------
TheEnd:
  END
```

Compile and execute the above Test.bas and then examine the contents of Test2.txt.

bx43.bas - Write Disk I/O

```
'  bx43.bas
'
  input$ = ""
  next$ = ""
  valuea = 0
  valueb% = 0
  valuec! = 0
  valued# = 0
'
  CLS
'
  PRINT "Opening Test File"
  OPEN "I", #1, "test.txt"
'
  OPEN "A", #2, "test2.txt"
'
Start:
  IF EOF(1) THEN
     GOTO Finish
  ENDIF
'
  INPUT#1, input$, next$, valuea, valueb%, valuec!, valued#
'
  IF input$="" THEN
     GOTO Start
  ENDIF
'
  PRINT input$, next$, valuea, valueb%, valuec!, valued#
  WRITE#2, input$, next$, valuea, valueb%, valuec!, valued#
'
  GOTO Start
'
Finish:
  CLOSE 1, 2
' -------------------------------------------
TheEnd:
  END
```

Is there going to be anything different in the result?. After executing it, examine Test2.txt.

bx44.bas - LINE INPUT#

LINE INPUT#:
============
Just as we can use the LINE INPUT command to enter an entire line of text from the keyboard, we can also us it to INPUT an entire line of data from a disk file. An entire line of data, up to 255 characters, terminated by a newline character, can be read in to a single string variable.

Before we try this, delete everything currently in Test.txt and copy this new informaion into Test.txt:

```
hello world, next string, 32000, 650000, 1.123, 3000000.123
hello , world, 2000, 50000, 0.123, 5000000.123
hello world, next string, 32000, 650000, 1.123, 3000000.123
```

Notice that there are no "quotes" surrounding either of the lines of text. Now try this Test.bas:

```
'  bx44.bas
'
  input$= ""
'
  CLS
'
  PRINT "Opening Test File"
  OPEN "I", #1, "test.txt"
'
Start:
  IF EOF(1) THEN
     GOTO Finish
  ENDIF
  LINE INPUT#1, input$
'
  IF input$="" THEN
     GOTO Start
  ENDIF
  PRINT input$
```

```
'
  GOTO Start
'
Finish:
  CLOSE 1, 2
' --------------------------------------------
TheEnd:
  END
```

bx45.bas - PRINT#

PRINT#:
=======
The function that mirrors LINE INPUT, oddly enough is not LINE WRITE, but, PRINT#. We are using the PRINT command and the hash-mark (#) indicates that we are PRINTing to a device.

Try this Test.bas and then examine Test2.txt:

```
'  bx45.bas
'
  input$ = ""
'
  CLS
'
  PRINT "Opening Test File"
  OPEN "I", #1, "test.txt"
  OPEN "O", #2, "test2.txt"
'
Start:
  IF EOF(1) THEN
     GOTO Finish
  ENDIF
  LINE INPUT#1, input$
'
  IF input$="" THEN
     GOTO Start
  ENDIF
  PRINT input$
  PRINT#2, input$
  GOTO Start
'
Finish:
  CLOSE 1, 2
```

```
' ------------------------------------------
TheEnd:
  END
```

bx46.bas - RANDOM I/O

```
RANDOM I/O:
===========
```

An additional method of disk I/O is called Random Access. Unlike sequential file access, which requires that disk file data be accessed sequentially, from beginning to end, Random access allows individual records to be written or read from a disk file in any desired order. Generally, in a random access file, records have a fixed length, with a predetermined number of characters or bytes. There by allowing records to be stored with each record having a known distance from the beginning of the file.

For instance, assume for a moment that a database will contain an unknown number of records, but, that each record will have a fixed length of one hundred ascii characters.

Example:

```
     first name:[             ] = 15 characters
      last name:[             ] = 15 characters
        address:[             ] = 30 characters
    information:[                 ] = 20 characters
    information:[                 ] = 20 characters
```

The record consists of the clients: name, address and information, for a total of 100 characters.

If this record were to be stored as record number one, then record number two would be stored beginning one hundred bytes away from the beginning of the file. You would say that record number two has an offset of one hundred bytes. Record number three would begin two hundred bytes from the start of the file and therefore have an offset of two hundred bytes.

A disk file of this description, opened for random access, would have the ability to read-in or write-out data using a disk buffer one hundred bytes in length. Reading in or writing out information one hundred bytes at a time, beginning at a calculated offset for a particular record.

Since a random access file has the ability to read or write individual records without disturbing the surrounding records, it need not be opened strictly for INPUT or for OUTPUT. Once a random access file is opened, it may be both read from and written to, at the same time.

Opening a random access file is no different from opening a sequential access file, with the exception that an "R" is used for the mode specifier and the filename/path is followed by the record length.

```
     i.e.:
          OPEN "R", #1, "test.txt", 50
```

The record length indicates the number of bytes the buffer will read or write for each record and will be used to calculate the offsets for each record.

There are five commands that are used when dealing with random access files;

```
          FIELD, LSET, RSET, PUT and GET.
```

FIELD: divides the I/O buffer into it's individual parts or strings of information that will comprise the record. Using the above example data:

```
      first name: = 15 characters
       last name: = 15 characters
         address: = 30 characters
     information: = 20 characters
     information: = 20 characters
```

the I/O buffer will be divided up like this:

```
                    I/O BUFFER
                   ------------
  [first$    ][last$     ][address$  ][info1$    ][info2$    ]
  ( 15 chars )( 15 chars )( 30 chars )( 20 chars )( 20 chars )
```

The usage for the FIELD command is:

```
        FIELD #1, 15 AS F$, 15 AS L$, 30 AS A$, 20 AS I1$,
(etc...)
```

LSET/RSET: are abbreviations for Left-Set and Right-Set.
In the event that the data contained in a particular string is too short, for instance if first$ only contains five characters and not all fifteen, LSET or RSET may be used to add padding, (blank spaces) to fill in the required length.

The difference between LSET and RSET is that: LSET pushes the existing string information to the far left of the string and adds any required padding to the right of the data.

RSET pushes the existing string information to the far right of the string and adds any required padding to the left of the data.

Example:

```
        First$ = "Fred"
        LSET [Fred...........]
        RSET [...........Fred]
```

LSET and RSET guarantee that each data string is the proper length when put into the buffer before writing it to the file.

Usage for LSET and RSET are:

```
        LSET F$ = first$
        LSET L$ = last$
        LSET A$ = address$
        RSET I1$ = info1$
        RSET I2$ = info2$
```

As a result of the LSET or RSET commands, the data is set to the proper lengths and placed into the I/O buffer.

PUT: is the command to write the buffer to the file.
The usage is:

```
        PUT 1,(record #)
        PUT 1, 99
        PUT 1, rec%   (where rec% is a long integer variable)
```

GET: is the command to read-in a file record into the buffer.
Usage is:

```
        GET 1,(record #)
        GET 1, 99
        GET 1, rec%   (where rec% is a long integer variable)
```

Here is an example of using the above random access commands:
(*** First, delete test.txt.***)

```
'  bx46.bas
'
   q$ = ""
   x$ = ""
   y$ = ""
   z$ = ""
   w$ = ""
   A$ = ""
   B$ = ""
   C$ = ""
   D$ = ""
   E$ = ""
'
   CLS
'
   q$ = "This"
   x$ = "is"
   y$ = "a"
   z$ = "test"
   w$ = "record"
'
   OPEN "R", #1, "test.txt", 50
   FIELD #1, 10 AS A$, 10 AS B$, 10 AS C$, 10 AS D$, 10 AS E$
   LSET A$ = q$
   LSET B$ = x$
   LSET C$ = y$
   LSET D$ = z$
   RSET E$ = w$
   PUT 1, 1
   CLOSE 1
' ------------------------------------------
TheEnd:
  END
```

Examine file: "test.txt".
Now change the record number in the PUT statement to read:

```
     PUT 1, 2
```

execute Test.bas and again, examine "test.txt".

bx47.bas - RANDOM I/O

```
'  bx47.bas
'
   q$ = ""
   x$ = ""
   y$ = ""
   z$ = ""
   w$ = ""
   A$ = ""
   B$ = ""
   C$ = ""
   D$ = ""
   E$ = ""
   rec% = 0
   xx = 0
'
    CLS
    '
    q$ = "This"
    x$ = "is"
    y$ = "a"
    z$ = "test"
    w$ = "record"
    rec% = 1
'
    OPEN "R", #1, "test.txt", 50
    FIELD #1, 10 AS A$, 10 AS B$, 10 AS C$, 10 AS D$, 10 AS E$
    FOR xx = 1 TO 5 STEP 1
        LSET A$ = q$
        LSET B$ = x$
        LSET C$ = y$
        LSET D$ = z$
        RSET E$ = w$
        '
        PUT 1, rec%
        rec% = rec% + 1
    NEXT xx
    CLOSE 1
'
    CLEAR
    DIM A$ = "", B$ = "", C$ = "", D$ = "", E$ = ""
'
    OPEN "R", #1, "test.txt", 50
    FIELD  1, 10 AS A$, 10 AS B$, 10 AS C$, 10 AS D$, 10 AS E$
    GET 1, 2
'
    PRINT A$
    PRINT B$
    PRINT C$
    PRINT D$
    PRINT E$
    CLOSE 1
' ------------------------------------------
TheEnd:
  END
```

bx48.bas - STRING I/O

```
I/O STRING FUNCTIONS:
=====================
```

As you've seen in the above, when reading from or writing to a random I/O file, the buffer is made up of fixed length strings. You may be wondering how then do we read or write numeric information, such as floating point numbers or the results of a calculation. This is accomplished with the "make" and "convert" functions.

The make functions are:

```
        MKD$() = make a double float into a string
        MKS$() = make a single float into a string
        MKI$() = make an integer into a string
```

and are used for writing data to a file.
The usage would be:

```
        A$ = MKD$(double#)
        B$ = MKS$(single!)
        C$ = MKI$(integer%) or (integer)
```

The convert functions are:

```
        CVD() = convert a string into a double float
        CVS() = convert a string into a single float
        CVI() = convert a string into an integer
```

The usage would be:

```
        double#  = CVD(A$)
        single!  = CVS(B$)
        integer% = CVI(C$)
```

```
'  test.bas version 48
```

```
'
   a# = 0
   b! = 0
   c% = 0
   d = 0
   rec% = 0
   ndx = 0
   q$ = ""
   x$ = ""
   y$ = ""
   z$ = ""
   w$ = ""
   A$ = ""
   B$ = ""
   C$ = ""
   D$ = ""
   E$ = ""
'
    CLS
    a#=1.012345
    b!=123.456
    c%=123456
    d = 12345
'
    q$ = MKI$(d)
    x$ = MKI$(c%)
    y$ = MKS$(b!)
    z$ = MKD$(a#)
    w$ = "end"
    rec% = 1
'
    OPEN "R", #1, "test.txt", 50
    FIELD #1, 10 AS A$, 10 AS B$, 10 AS C$, 10 AS D$, 10 AS E$
    FOR ndx = 1 TO 5 STEP 1
        LSET A$ = q$
        LSET B$ = x$
        LSET C$ = y$
        LSET D$ = z$
        RSET E$ = w$
        PUT 1, rec%
        rec% = rec% + 1
    NEXT ndx
    CLOSE 1
'
    CLEAR
    a# = 0
    b! = 0
    c% = 0
    d = 0
    A$ = ""
    B$ = ""
    C$ = ""
    D$ = ""
    E$ = ""
'
    OPEN "R", #1, "test.txt", 50
    FIELD  1, 10 AS A$, 10 AS B$, 10 AS C$, 10 AS D$, 10 AS E$
    GET 1, 2
'
    a# = CVD(D$)
    b! = CVS(C$)
    c% = CVI(B$)
    d =  CVI(A$)
    PRINT "a#="; a#
    PRINT "b!="; b!
    PRINT "c%="; c%
    PRINT "d ="; d
    CLOSE 1
' ------------------------------------------
TheEnd:
  END
```

```
Some other useful functions are:
        LOC() = returns the offset pointer within the current
                file
        LOF() = returns the length, in bytes, of the current file
        LEN() = returns the length of a string variable

Usages would be:
        pointer% = LOC(buffer)
            len% = LOF(buffer)
             len = LEN(mystring$)
```

bx49.bas - LOC FUNCTIONS

```
'  bx49.bas
'
   a# = 0
   b! = 0
   c% = 0
   d = 0
   pointer% = 0
   len% = 0
   A$ = ""
   B$ = ""
   C$ = ""
   D$ = ""
   E$ = ""
```

```
'
    CLS
'
    OPEN "R", #1, "test.txt", 50
    FIELD  1, 10 AS A$, 10 AS B$, 10 AS C$, 10 AS D$, 10 AS E$
    GET 1, 2
'
    pointer% = LOC(1)
    PRINT "offset="; pointer%
    len% = LOF(1)
    PRINT "length="; len%
    len% = LEN(A$)
    PRINT "len A$="; len%
'
    a# = CVD(D$)
    b! = CVS(C$)
    c% = CVI(B$)
    d =  CVI(A$)
    PRINT "a#="; a#
    PRINT "b!="; b!
    PRINT "c%="; c%
    PRINT "d ="; d
    CLOSE 1
' ------------------------------------------
TheEnd:
  END
```

bx50.bas - ARRAYS

```
ARRAYS:
=======
Multi-dimensional arrays can be created, resized and destroyed
with the following commands:

          DIM:        dimension an array
        REDIM:        redimension (resize) an array
        ERASE:        erase an array

The syntax for DIM is as follows:

        DIM MyLong%(10,10)       : creates a two dimensional
                                   array of long-integers

        DIM MyFloat!(10,10)      : creates a two dimensional
                                   array of single precision

        DIM MyDouble#(10,10)     : creates a two dimensional
```

```
                                    array of double precision
```

The syntax for REDIM is as follows:

```
        DIM MyValue%(1)           : creates a single dimension
                                    integer array

        REDIM MyValue%(2,2)       : redimensions array to a two
                                    dimension array
```

The syntax for ERASE is as follows:

```
        ERASE MyValue%            : removes array space from
                                     memory
```

***Note:** Arrays have global scope.

```
'  bx50.bas
'
'                 create single dimension array
  DIM MyArray%(5)
'
  CLS
'
  MyArray%(1) = 10
  MyArray%(2) = 20
  MyArray%(3) = 30
  MyArray%(4) = 40
  MyArray%(5) = 50
'
  PRINT MyArray%(1)
  PRINT MyArray%(2)
  PRINT MyArray%(3)
  PRINT MyArray%(4)
  PRINT MyArray%(5)
'
' ------------------------------------------
TheEnd:
  END
```

bx51.bas - ARRAYS

```
'  bx51.bas
```

```
'
'                 create single dimension array
  DIM MyArray%(5)
  value = 0
'
  CLS
'
  MyArray%(1) = 10
  MyArray%(2) = 20
  MyArray%(3) = 30
  MyArray%(4) = 40
  MyArray%(5) = 50
'
  PRINT MyArray%(1)
  PRINT MyArray%(2)
  PRINT MyArray%(3)
  PRINT MyArray%(4)
  PRINT MyArray%(5)
'
  value = MyArray%(1) + MyArray%(2)
'
  PRINT
  PRINT value
'
' -----------------------------------------
TheEnd:
  END
```

bx52.bas - ARRAYS

```
'  test.bas version 52
'
  value = 0
'                 create single dimension array
  DIM MyArray%(1)
'
  CLS
  MyArray%(1) = 99
'
  PRINT MyArray%(1)
  PRINT
'
'                 redimension single dimension array
  REDIM MyArray%(5)
'
  MyArray%(1) = 10
  MyArray%(2) = 20
  MyArray%(3) = 30
  MyArray%(4) = 40
  MyArray%(5) = 50
'
  PRINT MyArray%(1)
  PRINT MyArray%(2)
  PRINT MyArray%(3)
  PRINT MyArray%(4)
  PRINT MyArray%(5)
'
  value = MyArray%(1) + MyArray%(2)
'
  PRINT
  PRINT value
'
  ERASE MyArray%()
'
' ------------------------------------------
TheEnd:
  END
```

When using ERASE, how do you know the memory space is freed?
Try this next example:

bx53.bas - ERASE ARRAY

```
'  test.bas version 53
'
  DIM MyArray%(1)
'
  CLS
  MyArray%(1) = 99
'
  PRINT MyArray%(1)
  PRINT
'
  ERASE MyArray%()
'
'
  MyArray%(1) = 100
  PRINT MyArray%(1)
'
' ------------------------------------------
TheEnd:
  END
```

Now try these multi-dimensional examples:

bx54.bas – MULTI-DIMENSIONAL ARRAY

```
'  test.bas version 54
'
'                  create single dimension array
  DIM MyArray%(1)
'
'                  redimension to multi-dimensional array
  REDIM MyArray%(2,2)
'
  MyArray%(1,1) = 10
  MyArray%(1,2) = 20
  MyArray%(2,1) = 30
  MyArray%(2,2) = 40
'
  CLS
  PRINT MyArray%(1,1)
  PRINT MyArray%(1,2)
  PRINT MyArray%(2,1)
  PRINT MyArray%(2,2)
'
  ERASE MyArray%()
' ------------------------------------------
TheEnd:
  END
```

bx55.bas – MULTI-DIMENSIONAL ARRAY

```
'  bx55.bas
'
'                  create single dimension array
  DIM MyStrArry$(1)
'
'                  redimension to multi-dimensional array
  REDIM MyStrArry$(2,2)
'
  MyStrArry$(1,1) = "ten"
  MyStrArry$(1,2) = "twenty"
  MyStrArry$(2,1) = "thirty"
  MyStrArry$(2,2) = "forty"
```

```
'
  CLS
  PRINT MyStrArry$(1,1)
  PRINT MyStrArry$(1,2)
  PRINT MyStrArry$(2,1)
  PRINT MyStrArry$(2,2)
'
  ERASE MyStrArry$()
' ------------------------------------------
TheEnd:
  END
```

bx56.bas - WHILE/WEND

CONDITIONAL LOOPS:
==================
Besides IF/ELSEIF and FOR/NEXT statements Bxbasic also supports these conditional loops:

```
        WHILE/WEND

        WHILE   :       begins and tests a conditional loop
        WEND    :       end a conditional WHILE loop

and
        DO/WHILE

        DO      :       begins an unconditional loop
        WHILE   :       tests a conditional loop
```

WHILE/WEND:
===========
The basic WHILE/WEND loop begins with a test condition. If the condition tests TRUE then the loop is executed, up to the WEND and then re-tests the condition at the top of the loop.

Example:
```
        WHILE x < 10
             { do stuff...
               x = x + 1
             }
        WEND
```

In the above example, provided that variable "x" has a start value of less than 10, the loop will be executed. Variable "x" will be incremented during each pass through the loop. The loop will continue to execute as long as "x" is less than 10.

Let's replace the FOR/NEXT loops used in Example 23 with WHILE/WEND's in this example:

```
'  test.bas version 56
'
  DIM x = 0, y = 0
'
  CLS
  GOSUB Top
  GOSUB Center
  GOSUB Bottom
  GOTO TheEnd
'----------------------
Center:
  x = 1
  WHILE x <= 5
      PRINT "*";
      y = 1
      WHILE y <= 28
          PRINT " ";
          y = y + 1
      WEND
      PRINT "*":
      x = x + 1
  WEND
  RETURN
'----------------------
Top:
Bottom:
  x = 1
  WHILE x <= 30
      PRINT "*";
      x = x + 1
  WEND
  PRINT "":
  RETURN
'----------------------
TheEnd:
  END
```

bx57.bas - DO/WHILE

DO/WHILE:
=========
The DO/WHILE loop enters a loop or block of code prior to testing any conditions, therefore the loop/block is executed at least once everytime.

Example:
```
        DO
             { do stuff...
               x = x + 1
             }
        WHILE x < 10
```

In this case, "stuff" will be done, at least once, prior to WHILE testing for a TRUE condition.

DO/WHILE has not been thoroughly debugged and tested yet. It does work as a stand-alone loop. However, it does not work when used in a nested loop or with WHILE/WEND in a nested loop.

Here is a simple example to try:

```
'  bx57.bas
'
  x = 1
  DIM MyStr$(20)
'
  CLS
  MyStr$(1) = "At"
  MyStr$(2) = " present,"
  MyStr$(3) = " DO/WHILE"
  MyStr$(4) = " may"
  MyStr$(5) = " not"
  MyStr$(6) = " be"
  MyStr$(7) = " nested"
  MyStr$(8) = " or"
  MyStr$(9) = " used"
  MyStr$(10) = " in"
  MyStr$(11) = " conjunction"
  MyStr$(12) = " with"
  MyStr$(13) = " WHILE"
  MyStr$(14) = "/"
  MyStr$(15) = "WEND"
  MyStr$(16) = " loops."
'
  DO
```

```
    PRINT MyStr$(x);
    x = x + 1
  WHILE x <= 16
'
  PRINT
'
  ERASE MyStr$()
'
'-----------------------
TheEnd:
  END
```

bx58.bas - SWITCH/CASE

```
SWITCH/CASE:
============
```

The SWITCH/CASE pair, which is very similar to an IF/ELSEIF construct, allows for conditional program branching, based on a single integer value, tested by the SWITCH statement.

Example:

```
        SWITCH value
            CASE 1
                {do something...}
            CASE 2
                {do something else...}
            CASE 3
                {do something different...}
            DEFAULT
                {test failed, error handler...}
        ENDSWITCH
```

A similar IF statement would look like this:

```
        IF value = 1
            {do something...}
        ELSEIF value = 2
            {do something else...}
        ELSEIF value = 3
            {do something different...}
        ELSE
```

```
            {test failed, error handler...}
        ENDIF
```

In theory, the SWITCH/CASE block may be a little faster to execute, since it is designed to test if two integer values are equal.

```
'  bx58.bas
'
  IntVal = 0
  a$ = ""
'
  CLS
  PRINT "Press a number key,(from 1 to 5):"
  WHILE a$ = ""
      a$ = INKEY$
  WEND
'
  IntVal = CVI(a$)
  PRINT "The number you pressed was:";
'
  SWITCH IntVal
    CASE 1
      PRINT " One"
    CASE 2
      PRINT " Two"
    CASE 3
      PRINT " Three"
    CASE 4
      PRINT " Four"
    CASE 5
      PRINT " Five"
    DEFAULT
      PRINT " Oops!"
  ENDSWITCH
'
'----------------------
TheEnd:
  END
```