PEGASUS

EXTENDED BASIC

VER 1.0

XBASIC USERS NOTE

While this version of XBASIC will run without problems in a 4K Pegasus, program size is limited. It is adequate for small programs and most teaching applications but it is recommended that the 64K RAM expansion be fitted for large programing tasks.

1 INTRODUCTION

Pegasus Extended BASIC (XBASIC) is a very powerful interpreter supporting all standard BASIC features plus a wide range of special features unique to the Pegasus computer.

This manual assumes that the reader is familiar with BASIC programming and so detailed programming examples are not given (refer to the Pegasus BASIC manual if you are a beginner). Standard Pegasus keyboard conventions are supported (break, backspace, invert, etc.), and the usual control key functions may be directly invoked.

The following conventions are used in this manual:

- the statement or command being described is printed in capital letters.
- angle brackets (<>) enclose essential components of a statement.
- square brackets ([]) enclose optional components.

eg:

```
<essential element>
[optional element]
```

Note that the Network version of XBASIC also supports full disk I/O, virtual arrays and record I/O.

2 FUNDAMENTALS

2.0 LINES

Every line of an XBASIC program must begin with a LINE NUMBER. Lines may contain one or more statements separated by colons or backslashes and are terminated by a CARRIAGE RETURN. Valid line numbers are from 1 to 32767 inclusively and when run, an XBASIC program starts with the smallest line number and progresses to the largest (except when modified by a branch).

When writing programs, it is good practice to number lines in increments of 10 to allow room for additional ones to be inserted when the inevitable 'bug' becomes apparent.

eg:

```
30 INPUT "VELOCITY,TIME";V,T:PRINT "OK"
40 A=V/T:PRINT "ACCEL=";A
```

2.1 CONSTANTS

Numbers may be represented either as floating point or scientific notation in both positive and negative values. Six decimal digit precision is supported giving a range of:

```
  10E-38 to 10E38   positive
  0                 zero
-10E-38 to -10E38   negative
```

Numbers may be entered in any format (integer, floating point, or scientific), but XBASIC will automatically convert to scientific if they are too large or too small for its internal representation. In particular, this applies to any number which is over six digits long.

Expressions may also be used to represent numbers such as 2/3 to avoid having to type ".666666". Some examples of different number representations are:

```
floating point  257
                23.567

scientific notation 6.7E23
                    -5.7E-1

numerical expressions 1+3/7
                      22/7
```

Character string constants are also supported, and are most often used in PRINT and INPUT statements. String constants are defined by placing ASCII characters between single or double quotes.
eg:

```
"this IS a string"
```

2.2 VARIABLES

A variable may take on different values in the course of program execution. Two types of variable are supported, "real" and "character string". "Real" variables describe numbers, and consist of a single alphabetic letter or letter followed by one other letter or digit.
eg:

```
F, GT, Y6
```

"String" variables are defined by following any "real" variable name by a dollar sign ($). For example the above variables would be string variables if written as:

```
F$, GT$, Y6$
```

Either of the variable types may be subscripted (dimensioned) as described in the next section. Note that the same variable name may be used in a program as both real and string without conflict.

2.3 DIMENSIONING

Either of the two variable types may be subscripted using the DIM statement. This is accomplished by following the variable name with an integer enclosed in parentheses or two integers enclosed in parentheses and separated by a comma.
eg:

```
500 DIM R(3)
```

This will create four new variables R(0), R(1), R(2), and R(3).
When using a subscripted variable, it is written exactly as it appears in the DIM statement except that you may replace the subscript by any legal expression. Thus it may be used as shown above, or the subscript could be a variable. Two dimensional arrays are also supported:

```
50 DIM S(3,2)
```

This statement defines a matrix with the following elements:

```
S(0,0)  S(0,1)  S(0,2)
S(1,0)  S(1,1)  S(1,2)
S(2,0)  S(2,1)  S(2,2)
S(3,0)  S(3,1)  S(3,2)
```

Note that while a dimensioned array and a standard variable may share the same name, one and two dimensioned arrays cannot.

2.4 ASSIGNMENT

Variables may be assigned values by the LET, INPUT, or combination of the READ and DATA statements (see later sections for more detailed information).
examples:

```
20 LET W=2
```

assigns the value "2." to variable "W"

```
100 INPUT T
```

causes the computer to print a "?" and wait for the user to type in a number which is assigned to variable "T".

```
200 READ X
```

will assign a new value to X every time it is executed. The first time will assign to X the first piece of data from the first DATA statement in the program, the second time the second piece of the first DATA statement, etc. After all data has been read from one DATA statement, reading continues on to the next DATA statement, and continues through all DATA statements within the program. If a READ is attempted after the last piece of data from the last DATA statement has been used, then error 31 will be issued. A typical DATA statement would be:

```
600 DATA 2,5,6.08
```

2.5 OPERATORS

2.5.1 MATHEMATICAL OPERATORS

SYMBOL	EXAMPLE	MEANING
+	A + B	add A to B
-	B - A	subtract A from B
*	A * B	multiply A by B
/	A / B	divide A by B
^	A ^ B	raise A to the Bth power

When evaluating an expression containing several of these symbols, the following priority is used:

1. exponentiation
2. unary minus
3. multiplication & division
4. addition & subtraction

Thus when XBASIC is evaluating a mixed expression, each operator is applied firstly in the order shown above and thereafter (ie. equal priorities) from left to right as with normal arithmetic convention. Note that parentheses may be used to modify this order as they are always evaluated first. In the case of nested parentheses, evaluation starts with the innermost set and works its way out.

2.5.2 LOGICAL OPERATORS

Logical operators perform the desired operation on corresponding bits of numbers. eg:

```
 if A=(1100101111110110)
and B=(0111010111100100)

then NOT A=(0011010000001001)
   A AND B=(0100000111100100)
    A OR B=(1111111111110110)
```

Thus these are bit-by-bit operations yielding a single numerical result from one or two operands.

However logical operators have a totally different effect when they are used in an IF-THEN statement. In this instance, the expression is logically evaluated, not arithmetically, to yield a true or false result. An expression yielding a true result is assigned the value of "-1" while a false expression has the value "0".

```
40 IF S>2 AND S<5 THEN GOTO 10
```

This causes a branch to line 10 only if S is between 2 and 5 ie. if the expression "S>2 AND S<5" is true.

The following logical operators are available:

NOT eg NOT A - operates on integers to complement each bit of its binary representation.

AND eg A AND B - assigns each bit of the result a "1" if both of the corresponding bits of the arguments is a "1", else assigns the result bit a "0".

OR eg A OR B - assigns each bit of the result a "1" if either of the corresponding bits of the arguments is a "1", else assigns the result bit a "0".

2.5.3 RELATIONAL OPERATORS

These operators test the relationship between variables or constants:

SYMBOL	EXAMPLE	MEANING
=	A=B	A is equal to B
<>	A<>B	A is not equal to B
<	A<B	A is less than B
>	A>B	A is greater than B
<=	A<=B	A is less than or equal to B
>=	A>=B	A is greater than or equal to B

These are often combined with logical operators:

```
700 IF A=0 OR (F>9 AND S<>0) THEN GOTO 30
```

This will cause a branch to 30 if F is greater than 9 and S is not equal to 0, or if A is equal to 0. Otherwise the next line will be executed.

2.5.4 STRING OPERATORS

These consist of the concatenation operator "+" and the relational operators. The "+" may be used to join (concatenate) two strings to form a new string, and the relational operators indicate alphabetic sequence within a string. If one string is greater that another, this implies it would appear after the other if they were sorted into alphabetical order. In this type of comparison, trailing blanks are ignored and if the two strings are of unequal length, the shorter string is padded with trailing blanks to make it equal for comparison purposes. A null string (zero length) is considered completely blank as is a string of all spaces for comparison purposes. All the standard arithmetic relational operators may be used in connection with strings.

2.5.5 OPERATOR PRECEDENCE

The overall operator precedece is shown as follows (highest to lowest), and operators of equal precedence are evaluated from left to right.

() expressions enclosed in parentheses
^ exponentiation
- unary minus
* / multiplication and division
+-- addition and subtraction
relational operators
NOT
AND
OR

2.6 MODE

XBASIC can function in program or immediate modes. In the program mode XBASIC executes lines progressing from the smallest to the largest statement numbers. However the immediate mode does not have line numbers, and any command given is executed immediately it is entered.
eg if you typed:

```
300 PRINT S
```

it would be treated as part of a program and is merely stored for future use.

while:

```
PRINT S
```

would be executed immediately because there is no line number.

2.7 REMARKS

It is good practice to use remarks throughout your program in order to make it easier for you and others to follow. After a "REM" all other text on that line is ignored by the XBASIC interpreter, thus allowing comments to be inserted.
eg:

```
30 REM this is a silly remark
```

Note that this is very wasteful of memory space and so should be used cautiously on a 4K Pegasus.

3 COMMANDS

Commands are not used in the actual code of XBASIC programs but are direct instructions to the computer. They may be used any time that XBASIC is in the command mode (after a "READY" prompt) and the action is taken immediately.

CLEAR - resets all program variables to 0 (note that it is automatically performed when RUN is executed).

CLS - clears the video screen and moves the cursor to the top left hand corner.

CONT - restarts a program after it has been stopped by a STOP command or the "break" key. The command cannot be used after an error or if more program lines have been entered. Program execution continues at the first statement following the stop but if the break was used at an input statement, execution resumes at that statement.

EXIT - causes a return to the Pegasus menu.

LIST - may be used to display lines of your program and may be terminated with the break key. Examples of its use are as follows:

```
LIST  (lists the entire program)
LIST 10  (lists line 10)
LIST 20-45  (list lines 20 through 45)
```

LINES - when followed by a number between 1 and 16 sets the number of lines to be displayed on the screen. Note that the less lines the faster a program will execute.

NEW - deletes the current program allowing you to type in a "new" one.

RAWOFF - returns to normal control character operation (see RAWON).

RAWON - sets a special flag which causes all subsequent control characters to be echoed to the screen as special printing characters without being executed.

RUN - causes the computer to begin execution of the current program. All variables are initialized and all DATA statements are restored.

SETHR - followed by a value setshe internal time in hours.

SETMIN - followed by a value sets the internal time in minutes.

SETSEC - followed by a value sets the internal time in seconds.

TLOAD - permits previously saved programs to be loaded from cassette tape in the normal Pegasus manner.

TROFF - turns off the trace mode (see TRON).

TRON - turns on the trace mode causing each line number to be printed out (in brackets) as it is executed. This is very usefull for debugging programs.

TSAVE - saves XBASIC programs on cassette tape in the normal Pegasus manner.

Note that LINES, CLS, RAWON, RAWOFF, SETHR, SETMIN, and SETSEC may also be used as program statements if required.

4 STATEMENTS

4.1 ASSIGNMENT

DATA - <line number> DATA <number> [,<number>,<number>,...]
" DATA <string> [,<string>,<string>,...]

This statement specifies information that will be read in by the program each time a READ statement is encountered. Data is read from left to right and once all of one DATA statement has been read, reading will automatically continue from the next (by line number) DATA statement. Note that the DATA statement cannot appear in multiple statement lines.
eg:

```
100 DATA 27.6,0,-1E-5
110 DATA fred,HARRY,Bill
120 DATA "345 P","13,500"
```

LET - <line number> LET <variable>=<expression>

This is used to assign a value to a variable, which may be a constant or complex expression. Note that LET is optional and may be safely left out of any assignment statement.
eg:

```
30 LET I=3.56
40 LET F$="Hello"
50 LET W(R,S)=6
60 K=H3/(5*U)
```

READ - <line number> READ <variable>
[,<variable>,<variable>,...]

This is used to read data sequentially from DATA statements. Note that each argument of the READ statement is assigned a new piece of data in the fashion previously described. Any attempt to read beyond the last DATA statement will result in error 31.
eg:

```
250 READ E,A$
```

RESTORE - <line number> RESTORE

When a program is run the first read comes from the first DATA statement and progresses sequentially through the available data as further reads are executed. A RESTORE resets this operation so that the next read is forced to operate on the first item of the first DATA statement. Sequential operation then continues as before.
eg:

```
500 RESTORE
```

4.2 TRANSFER of CONTROL

GOSUB - <line number> GOSUB <line number>

This causes program execution to continue at the subroutine line number specified after "GOSUB". However unlike a GOTO, all subroutines end with a RETURN statement which causes program execution to return to the statement after the GOSUB instruction which called the subroutine.
eg:

```
40 GOSUB 500
50 REM it will return to here
```

This causes the program to go to a subroutine at line 500, and the next RETURN encountered will return control to line 50
eg:

```
500 REM this is a subroutine
510 RETURN : REM return to line 50
```

GOTO - <line number> GOTO <line number>

This simply causes program execution to branch to the line number specified after the "GOTO".
eg:

```
30 GOTO 10
```

ON GOSUB - <line number> ON <expression> GOSUB <list of line numbers>

This statement allows the calling of one of a list of possible subroutines, chosen by the value of "<expression>". The expression is evaluated and truncated to an integer, and then used as a pointer into the list of line numbers specified which are considered to be 1,2,3... etc. Hence if the expression evaluates to 4 then the 4th line number will be used for the subroutine call. Note that an error will occur if the expression evaluates to a number which does not have a corresponding entry in the list of line numbers (ie -3). The ON GOSUB statement should be the last statement on any line.
eg:

```
40 ON R GOSUB 10,20,30
```

ON GOTO - <line number> ON <expression> GOTO <list of line numbers>

This operates exactly like ON GOSUB as described previously, except that a GOTO type branch occurs instead of a subroutine call. Note that no statements should follow the ON GOTO on lines containing multiple statements, as they would never be executed.
eg:

```
50 ON Y GOTO 300,400,500
```

ON ERROR GOTO - <line number> ON ERROR GOTO [<line number>]

This allows the user control over errors that may occur during the course of program execution. The ON ERROR statement tells XBASIC where to GOTO if an error occurs. Thus the programmer can create special routines to handle particular fault conditions that may occur when his program is running. Note that this will also work on syntax errors and so should only be implemented when the program is free of this type of error, otherwise debugging can become confusing (although in some instances this may be used to advantage).

RESUME - <line number> RESUME <line number>

This is used to return control to the main program after an error routine has been executed (refer section on error handling for further information).

RETURN - <line number> RETURN

This tells the computer that it has reached the end of a subroutine and should return to the calling routine (see GOSUB).

4.3 CONDITIONAL

IF GOTO - <line number> IF <expression> GOTO <line number>

If the expression is true then the program branches to the specified line number. If the expression is false then the program continues on with the next program line (not the next statement in a multiple statement line).
eg:

```
30 IF V<>0 GOTO 100
500 IF G=1 AND K>9 GOTO 870
```

IF THEN - <line number> IF <expression> THEN <line number>
" IF " THEN <statement>

This functions in a similar way to IF GOTO except that instead of having to branch to another line number, a statement may be executed instead. If the expression is true then the statement will be executed, otherwise control will be passed on to the next sequential line and the statement ignored. Note that multiple statements (separated by ":") may be incorporated, and will either all be executed or all ignored depending upon the validity of the expression.
eg:

```
20 IF G>7 THEN 70
30 IF T1=0 THEN PRINT "OK"
40 IF Y<>45 THEN PRINT "OK":Y=0
```

IF THEN ELSE -
<line number> IF <expression> THEN <line number> ELSE <line number>
" IF " THEN <statement> ELSE <statement>

This acts the same way as IF THEN except that if the expression is false the ELSE statement is executed. Thus one or other of the two statements (line numbers) is always executed. Note that this and all preceeding conditionals may be nested.

4.4 INPUT/OUTPUT

INPUT - <line number> INPUT ["string";] <variable list>

When executed, this statement prints out the character string enclosed in quotes followed by a question mark. It then waits for an input (or prompts for multiple inputs) from the keyboard, and assigns those values to the variable list as specified. Note that if quotes are required to be printed within the string, then single quotes may be used to surround the whole message
eg:

```
INPUT 'Type in "feeling" '; S
```

As long as there are unassigned items in the variable list; XBASIC will continue prompting for input with a "?". Note that every input from the keyboard must be terminated with a carriage return.

INPUT LINE - <line number> INPUT LINE <string variable name>

This statement is used to input a whole line from the keyboard into a string variable. All quotes and other punctuation are accepted. Note that no text string may be output as part of this statement and only one variable name may be listed.
eg:

```
10 INPUT LINE R$
```

PRINT - <line number> PRINT [[x,y]] [variable, string; ...,]

The print statement has many options and only the word PRINT is always required (on its own it just causes a CR LF). Any arbitrary combination of strings, variables and expressions may be printed. If the statement ends with a "," or ";", a CR LF is not performed, allowing formating of the output across the screen

XBASIC divides each output line into 5 fields of 8 (or switches automatically to 16 if the output will not fit) character positions each. When arguments are separated by commas, this will cause each argument to be printed at the beginning of the next field. Note that several commas may be typed in succession to skip several fields.

When arguments are separated by a semicolon, they will be printed immediately following each other without gaps. Note that numbers are automatically printed with a single trailing and leading space (if not negative).

In addition, exact cursor positioning on the video screen is possible using the optional [x,y] argument, where x is the x coordinate (0-31) and y is the y coordinate (0-15) at which the first character will be placed. Note that this feature may not be used if outputing to the printer (see later).
eg:

```
10 PRINT
20 PRINT "hello"
30 PRINT "LOOP=";3
40 PRINT "ANSWER =",A/E+4
50 PRINT [3,7] Z$,
```

4.5 LOOPS

FOR - <line number> FOR <variable>=<expr1> TO <expr2> [STEP<expr3>]

This statement will cause all of the following instructions to be executed until a NEXT statement is reached. Execution then continues in a loop between the FOR and the NEXT statements until <variable> is equal to <expr2>. Each time round the loop <variable> is automatically incremented by <expr3> (which defaults to 1 if not specified), and <variable> begins with a the value of <expr1>.

FOR statements may be nested but they cannot use the same index variable, and they may be exited prematurely with a GOTO (note that they must not be entered in this way). Otherwise, once the <variable>=<expr2> condition is reached, execution continues with the program line following the NEXT statement.

It is advisable to use integer values in the loop count otherwise the situation may result where XBASIC's internal rounding off error prevents <variable> and <expr2> from ever matching.

eg:

```
30 FOR E=1 TO 7 STEP 2
40 REM this will loop 3 times for E=1,3,5
50 NEXT E
60 REM program will resume here when E=7
```

NEXT - <line number> NEXT <variable>

This is only used with the previously described FOR statement to terminate a FOR NEXT loop. NEXT always causes a branch back to the corresponding FOR statement, and <variable> must be the same name as the one used in the FOR statement.

4.6 TERMINATION

END - <line number> END

The END statement causes the program to terminate and it cannot be restarted with the CONTINUE command. It is optional (without it "end" occurs automatically after the last program line is reached), but when used it may be placed anywhere in the program.

STOP - <line number> STOP

When this statement is executed, the program is terminated and a message printed. However, unlike END the program may now be restarted with CONTINUE.

4.7 MISCELLANEOUS

DEF -
<line number> DEF FN<variable>[(dummy variable)]=<expresion>

The programmer may define single line functions with this statement. Although the same function may be redefined many times throughout the program, only the latest definition is used when the function is encountered. It may contain only one argument and the dummy variable used to represent this in the definition has no meaning elsewhere in the program.

Any valid variable name preceeded by "FN" is allowed (except for string variables), and once defined, a function may be called anywhere in the remainder of the program just by the appearance of its name. At that time the function is evaluated as per its definition, and that value is assigned to it as if it was an ordinary variable.
eg:

```
10 DEF FNMT(V)=10*V
```

defines a function FNMT which whenever referenced with a variable will return with the value of that variable multiplied by 10.
eg:

```
60 F=9
65 Y=FNMT(F)/3
```

would return a function value of 90 and thus Y would be assigned the value of 30.

DIM - <line number> DIM <var1>(n) [,<var2> (m),...]
" DIM -" (k,l) " (m,n)

This was described in detail previously, and reserves memory for any arrays that are to be used in the following program.
eg:

```
20 DIM A(5),T(8,9),S$(2)
```

POKE - <line number> POKE <address>,<data>

This statement allows direct output to memory (or ports). Note that address and data must be given in decimal form and that extreme care must be taken to avoid poking incorrect data into crucial areas of the computer's memory map.
eg:

```
300 POKE 20000,67
400 POKE B,U
```

REM - <line number> REM [message ....]

As shown previously, this statement simply allows messages or comments to be inserted into your program to make it more easily understood.

5 SUPPLIED FUNCTIONS

5.1 MATHEMATICAL

EXP(X) - returns the value of e raised to the power of x, where e is the base of natural logs.

LOG(X) - returns the value of the natural log (base e) of X. Note if logs of other bases are required the following formula may be used:

LOG of X to the base B = LOG(X)/LOG(B)

SQR(X) - returns the value of the square root of X.

5.2 TRIGNOMETRIC

ATN(X) - returns the value of the arctangent of X, in radians.

COS(X) - returns the value of the cosine of X, and assumes that X is in radians.

SIN(X) - returns the value of the sine of X and assumes that X is in radians.

TAN(X) - returns the value of the tangent of X and asssumes that X is in radians.

5.3 CHARACTER

ASC(X$) - The value returned by this function is the ASCII numeric value of the first character in the string X$. Note that zero will be returned if the argument is the null string.

CHR$(J) - The value returned by this function is a single ASCII character string whose numeric value is that of the argument J. Note that J must be in the range 0 <= J <= 255

HEX(X$) - The value returned by this function is the decimal equivilant of the hexadecimal character string X$.

INKEY$ - The value returned by this function is a single ASCII character string corresponding to the value of any key that it pressed on the keyboard. If no key is pressed then the null string will be returned.

LEFT$(X$,J) - This function returns the string that is the J left most characters of the string X$. Note that J must be positive and less that 32767.

LEN(X$) - This function returns the number of characters in the string X$. Note that all characters including spaces and non printing control characters are counted.

MID$(X$,J) - This function returns the portion of the string X$ that begins at position J and includes everything until the end of the string. Note that J must be between 0 and 32776.

MID$(X$,I,J) - This function works in the same way as MID$ with only two arguments except that the returned string only includes J characters.

RAW(J) - This function returns the raw mode character whose numeric value is J.

RIGHT$(X$,J) - This function returns a character string that is the rightmost J elements of string X$. Note that if J is greater than or equal to the length of the string, then the entire string will be returned.

STR$(X) - This function returns a character string that represents the numerical expression X. It is created with a leading space or minus sign and a trailing space.

VAL(X$) - This function does the opposite of STR$ and creates a numerical value out of the string X$. Note that 0 will be returned if X$ does not represent a valid numerical value.

5.4 INPUT/OUTPUT

PEEK(J) - This functions in the opposite way to POKE. It returns a value which is the contents of memory location J (decimal). Note that it is normally used with the HEX function and returns a number between 0 and 255.
eg:

```
100 R=PEEK(HEX("C000"))
```

POS(J) - This function returns the current cursor position (x coordinate only) on the screen. Note that numbering starts at 0.

SPC(J) - This function is used in a PRINT statement and causes J spaces to be printed.

TAB(J) - This function is used in a PRINT statement and moves the cursor to column J of the screen (or printer). Note that if the cursor is already past this point then no action will be taken.

5.5 MISCELLANEOUS

ABS(X) - This function returns the absolute value of X.

FRE(0) - This function returns the amount of free memory left to the programmer

HRS - Returns the value of the internal "hours" counter.

INT(X) - This function returns the largest integer which is not greater than X.

MINS - Returns the value of the internal "minutes" counter.

PI - Returns the value of "PI"

PTR(variable name) - This function returns the address of the named variable. Floating point variables are stored as 4 bytes beginning at the specified address. The first 3 bytes are mantissa (stored as sign plus magnitude with sign in the msb position), and the 4th is exponent (base 2, biased by hex 80). The mantissa is maintained in hidden bit normalized format.

If the argument is a string variable, the address returned is that of a 4 byte descriptor. The first 2 bytes are the actual address of the string, and the second 2 bytes are the string length.

RND(X) - This function returns a value between 0 and 1 which may be used to generate random numbers in a given range as follows:

Random number = (ML-MS)*RND(0)+MS

Where the resulting random number will be in the range MS<=RND<=ML.

The argument X has the following function:

X<0 - causes a new sequence of random numbers to be started, a different sequence for each different value of X.

X=0 - causes the function to generate the next random number in the sequence.

X>0 - causes the function to return the last random number generated.

SECS - Returns the value of the internal "seconds" counter.

SGN(X) - This function returns a "1" if the argument is greater that zero, zero if the argument is zero, and a "-1" if it is negative.

6 OUTPUT TO A PRINTER

All normal output from a PRINT statement may in turn be redirected to a printer (usual Pegasus interface), by specifying channel 1.
eg:

```
30 PRINT #1, "this will print on the printer"
```

Programs may be listed to the printer by the command:

```
LIST #1
```

7 ERROR HANDLING

As described previously, the ON ERROR GOTO - RESUME statements allow the user to specify a special routine which will be executed should an error occur in his program, instead of the usual error message. Of particular importance is the way in which RESUME is handled.

```
<line number> RESUME [<line number>]
```

If a line number is specified after RESUME then XBASIC will restart program execution at that line. Otherwise XBASIC will resume by re-executing the line that caused the error. This can cause problems if there were multiple statements in that line, as all will be executed, and also if the error condition persists, as the program will hang up in a permanent loop.

To avoid these difficulties, two methods are commonly employed. Firstly a known "safe" line number may be specified in the RESUME statement to ensure that the program returns to a suitable place. Secondly, special error handling routines may be created using the ERR and ERL system variables. These two variables are automatically updated by XBASIC whenever an error occurs and contain the error number (ERR) and line at which it occured (ERL). Thus the programmer's error handling routine, and subsequent program resumption, may be set up to act differently depending on the type and location of the offending error.

Note that the ON ERROR GOTO feature may be selectively disabled (and re-enabled) during the course of program execution. Re-enabling is accomplished by specifying the ON ERROR GOTO with a valid line number to go to. Disabling is accomplished by specifying a line number of "0". If the ON ERROR feature is disabled during an error program (but before the RESUME), then the normal XBASIC error message will be printed.
example:

```
10 ON ERROR GOTO 500
20 INPUT "enter a number",Z
30 PRINT "its square root is ",SQR(Z)
40 GOTO 20

500 IF ERR<>30 THEN ON ERROR GOTO 0
510 PRINT "enter a number!!"
520 RESUME
```

8 THE USR FUNCTION

This function allows the programmer to call a machine language subroutine from within the XBASIC program. It is normally used where a special routine is required that cannot be easily created in XBASIC ie where speed may be critical. Note that space may be saved for the machine code routine by setting the memory end pointer (see Pegasus Monitor documentation) below its usual value prior to entering XBASIC.

When the USR function is encountered in an expression, the argument is evaluated, converted into a 16 bit 2's complement integer and placed at location MEMEND-4. Next, an address is obtained from MEMEND -2 and this is used to tell XBASIC where the required machine code subroutine is located. If these bytes are 0 (to which they are automatically set on power up), then an error will be issued. If they are non-zero then a JSR will be executed to the address specified. Note that this address may be set manually (from the monitor) or by using the POKE statement.

If it is desired to pass a parameter from the users routine back to XBASIC, this may be left in 16 bit 2's complement form at MEMEND-4. This is automatically returned as the value of the USR statement. Note that the user subroutine must always end in an RTS, and must return with the U, S, and Y registers unchanged.

eg:

assuming that MEMEND has been set to $AFFF and space is reserved for the user routine at $B000

```
100 POKE HEX("AFFD"),HEX("B0")
110 POKE HEX("AFFE"),HEX("00")
120 Z=USR(4)
```

the corresponding machine code program would reside at $B000

```
ORG $B000      .origin
LDX $AFFB      .get passed parameter
****           .user code here
LDX #VALUE     .VALUE=return parameter
STX $AFFB      .save parameter for XBASIC
RTS            .must end with this
```

Although there is provision for only one USR routine in XBASIC, multiple machine code routines may be invoked by changing the USR address vector prior to calling it via a POKE.

SUMMARY OF STATEMENTS AND COMMANDS

STATEMENTS:

DATA
DEF
DIM
END
FOR
GET
GOSUB
GOTO
IF
INPUT
INPUT LINE
LET
NEXT
ON ERROR
ON GOSUB
ON GOTO
POKE
PRINT
READ
REM
RESUME
RETURN
STOP

COMMANDS:

CLEAR
CLS
CONT
EXIT
LINES
LIST
NEW
RAWOFF
RAWON
RUN
SETHR
SETMIN
SETSEC
TLOAD
TROFF
TRON
TSAVE

FUNCTIONS:

ABS
ATN
COS
ERR
HEX
INKEY$
LEFT$
LOG
MINS
PI
PTR
RIGHT$
SECS
SIN
SQR
TAB
VAL
ASC
CHR$
ERL
EXP
HRS
INT
LEN
MID$
PEEK
POS
RAW
RND
SGN
SPC
STR$
TAN

SYNTAX AND COMPUTATIONAL ERRORS

NUMBER	MEANING
50	UNRECOGNIZABLE STATEMENT
51	ILLEGAL CHARACTER IN LINE
52	SYNTAX ERROR
53	ILLEGAL LINE TERMINATION
54	LINE NUMBER 0 NOT ALLOWED
55	UNBALANCED PARENTHESIS
56	ILLEGAL FUNCTION REFERENCE
57	MISSING QUOTE IN STRING CONSTANT
58	MISSING "THEN" IN AN "IF" STATEMENT
60	BAD LINE REFERENCE IN "GOTO" OR "GOSUB"
61	RETURN WITHOUT "GOSUB"
62	"FOR-NEXT" NEST ERROR
63	CAN'T CONTINUE
64	SOURCE NOT PRESENT
65	BAD FILE - WON'T LOAD
66	"RESUME" NOT IN ERROR ROUTINE
72	MIXED MODE ERROR
73	ILLEGAL EXPRESSION
74	ARGUMENT OUT OF RANGE
75	ARGUMENT OUT OF RANGE
76	ILLEGAL VARIABLE TYPE
77	ARRAY REFERENCE OUT OF RANGE
78	UNDIMENSIONED ARRAY REFERENCE
80	MEMORY OVERFLOW
81	ARRAY OVERFLOW
90	UNDEFINED USER FUNCTION
91	UNDEFINED USER CALL
94	BAD STRING LENGTH SPECIFIED
95	MULTIPLE VIRTUAL ARRAY REFERENCE
100	EXPRESSION TOO COMPLEX
101	OVERFLOW OR UNDERFLOW IN FLOATING POINT OP
102	ARGUMENT TOO LARGE
103	DIVISION BY ZERO
104	NUMBER TOO LARGE TO CONVERT TO INTEGER
105	NEGATIVE OR ZERO ARGUMENT FOR "LOG"
106	CONVERSION ERROR IN INTEGER "INPUT"
107	IMAGINARY SQUARE ROOT
108	CONVERSION ERROR (NUMBER TOO LARGE)
255	ILLEGAL TOKEN ENCOUNTERED
30	DATA TYPE MISMATCH
31	OUT OF DATA IN "READ"
32	BAD ARGUMENT IN "ON"STATEMENT
1	DEVICE NOT READY
59	LINES 0 NOT ALLOWED