QuickCalc User Guide.

Number Representation, Assignment, and Conversion

Variables

"Double" (or "DOUBLE") floating-point variables (approx. 16 significant digits, range: approx. $\pm 10^{308}$

The actual range of DOUBLE floating-point variables is:

maximum: ±1.7976931348623157e+308, Anything larger is considered "infinite"

minimum: ±4.9406564584124654e-324. Anything smaller becomes zero. This is the smallest number that can be represented in DOUBLE floating point. It has only one bit of significance, and equals (in hex) .0000000000001 x 2^{-1023}.

minimum with full precision (all digits significant): ±2.2250738585072014e-308. Numbers smaller than this lose precision.

"Long" (or "LONGMATH") floating-point (or integer) variables (length up to 1 million, limited by machine memory, range: approx. $\pm 10^{1,000,000}$.

String variables and arrays, which may hold number strings in certain cases, described below.

Constants

Built-in constants like "**PI**" (DOUBLE) and "**LONGPI**" (LONGMATH).

Number strings, which are character strings containing digits in the valid BASIC format. Example: [**±**][dd...dd][**.**][dd…dd][[**e**|**E**][**±**]dd…dd]. The number string must be able to convert correctly into the format determined by its ultimate use.

Usage

Variable names may appear in your program wherever a number is desired. The current value of that variable will be used. If no value has been assigned to that variable, it is assumed to have a zero value.

Note: Variable names may also be referenced indirectly (by string expressions) using the **@**(*string-expression*) function. See "*Indirect Name Reference*", below.

Number strings may be coded into your program (without quotes) in expressions, as subscripts, as function parameters, or anywhere as number is required. In

general, number strings, or constants, may be used interchangeably with variables, except that you cannot assign a value to a number string, and statement numbers may not be variables.

Number strings may also appear in **DATA** statements, and in data read in through **INPUT** and **INPUT #** statements. (see "Reading, Writing, and Printing Long Numbers" below.)

Assignment

A number string may be assigned to a variable.

It will be converted, if possible, to match the type of variable to which it is being assigned. An error will be generated if the number cannot be converted.

Number strings are not converted until they are used. They are stored in their string format until assigned to a variable or used as a parameter to a function or in some other way. In this way, precision is not lost if the number will ultimately be used as a LONGMATH, and time is not wasted converting it into a LONGMATH if it will be used as a DOUBLE. (See "Conversion")

Number strings may be assigned to string variables, if desired, in which case they are not converted.

One variable may be assigned to another.

It will be converted, if possible, to match the type of variable to which it is being assigned. An error will be generated if the number cannot be converted.

Numeric variables may be assigned to string variables or string arrays (described below).

String variables containing valid numeric strings may be assigned to DOUBLE variables.

If a string is not a valid numeric string, the **VAL ( )** function may be used to extract the numeric portion from the beginning of the string.

String variables (and constants) may NOT be assigned to LONGMATH variables (exception: see the **LONG** ( ) function, below).

The result of an expression evaluation or function will be either LONGMATH, DOUBLE, or string, and may be assigned to a variable as described above.

Conversion

In any arithmetic operation, numbers will be automatically converted to match each other, as follows:

If one operand is LONGMATH and the other is DOUBLE, the DOUBLE will be converted to LONGMATH, the operation will be performed using long arithmetic, and the result will be LONGMATH.

If one operand is LONGMATH or DOUBLE and the other is a number string, the number string will be converted (if possible) to match the other operand. If it can't be converted into a DOUBLE, then both operands will be converted to LONGMATH.

If both operands are number strings, they will be both converted to DOUBLE, if possible. Otherwise, they will both be converted into LONGMATH. Numbers longer than 16 digits, greater than ±1.7976931348623157e+308 or smaller than 2.225073858507201e-308 will automatically be converted to LONGMATH to avoid loss of precision.

A string or string variable which contains a valid number string may be used anywhere a number string is valid (in expressions, parameters, subscripts, etc.) For example, *a* = **SQR** ("3") is valid. The string will be interpreted as a number string and converted to a DOUBLE value of 3.

a = **SQR** ("ABC") is not valid because "ABC" cannot be converted to a number.

An exception is the **+** function. For strings, this signifies concatenation ("AB" **+** "CD" = "ABCD"). You can't mix strings and numbers in this case.

Number strings with exponents too large for DOUBLE, or containing more than 16 digits, will automatically be converted into LONGMATH.

Functions with more than one parameter will have the parameters converted to match each other (if necessary), as described above.

The results of one operation will affect the next operation in an expression. Since expressions are evaluated from left-to-right, and conversions are performed as needed, the way you order your expression may affect the result.

Long (LONGMATH) vs. short (DOUBLE) Functions

Mathematical functions have long and short versions, depending on the type of arguments they are passed. If the argument or arguments is/are LONGMATH, the long version of the function is called, resulting in a LONGMATH being returned. If there are more than one argument and they are different types, the DOUBLE will be converted up, if necessary, to match the LONGMATH.

Functions which have long and short versions are:

sqr (*x*)	half (*x*)	sgn (*x*)	sin (*x*)	cos (*x*)
tan (*x*)	cot (*x*)	atn (*x*)	acot (*x*)	log (*x*)
log10 (*x*)	exp (*x*)	exp10 (*x*)	abs (*x*)	fix (*x*)
int (*x*)	cint (*x*)	mod (*x*, *y*)	str$ (*x* [, *f*$])	

Functions which expect short arguments, but will use long values and convert them, are:

hex$ (*x*, *y*)	short (*x*)	open (..*x*..)	chr$(*x*)
string$ (*x*,*y*)	space$ (*x*)	tab (*x*)	instr ([*x*,] *a*$, *b*$)
left$ (*a*$, *x*)	right$ (*a*$, *x*)	mid$(*a*$, *x*, *y*)	tab (*x*)

Statements which expect short arguments, but will use long values and convert them:

color (*x*)	float (*x*)	open (..*x*..)	close (*x*,..)	mid$ (*a*$, *x*, *y*)

LONG (*x*) accepts anything (number strings, numbers, strings and string arrays) and returns a LONGMATH.

VAL (*x*$) returns a numeric string, which can be assigned to a numeric variable (DOUBLE or LONGMATH) or used in a numeric expression. If it is assigned to a string variable or printed directly, it will be treated as a string containing the numeric part of *x*$.

The **FOR** statement will use either a DOUBLE or LONGMATH as the control variable. The expressions for the initial value and the **TO** and **STEP** values will be converted to match the control variable. It is far more efficient (fast) to use DOUBLEs than LONGMATHs if a long control variable is not necessary.

The **WHILE** statement expects an expression. The expression is considered TRUE if it results in a number which is not zero, a string which is not empty, or a

number string which does not convert to zero. The same is true for the **IF** statement.

Note: All variables and arrays are, by default, assumed to be double, and most functions default to DOUBLE math, unless one of the arguments is LONGMATH (see above). You can, optionally, choose to run your program with LONGMATH numbers and variables only. (See "Working With Long Numbers".)

Long number representation as a string or string array.

In order to represent long (LONGMATH) numbers as character strings, it is allowable to assign a variable (or the result of an expression or function) to a string (*if it is less than 256 characters long*), or to a **string array** which is dimensioned to have enough elements to hold the entire long number.

The string array is specified as *stringarray*$ and is used as follows:

stringarray$ = *expression which evaluates to a [long] number*

A DOUBLE variable may be assigned to a string by first converting it into a LONGMATH value (using the **LONG ( )** function) and then assigning it to the string. Caution: the resulting LONGMATH number could be as long as the floating-point length, which may be too long for a string variable. You may, of course, also use the **STR$** function.

You may NOT perform arithmetic on the resulting strings, except

LONG (*string*$) or **LONG** (*stringarray*$) converts the string or string array into a LONGMATH number which can be used wherever a number is desired.

You may use string operations on the strings in the string array. Be careful not to invalidate the number format if you plan on using the number in a subsequent arithmetic operation or function.

Reading, Writing, and Printing Long Numbers.

You may code a LONGMATH number value into a **DATA** statement, enter it as a response to an **INPUT** statement, or type it into a file which will be read by an **INPUT #** statement. The limit on length is 255 characters, including signs, decimals, and exponent. This is in the form of a number string, and is not enclosed in quotes.

LONGMATH values may be printed with the **PRINT** statement. They will be formatted into a long string and printed without quotes on as many lines as

needed, with 80 characters per line. If printed to a file with **PRINT #**, the lines will be as long as the logical record length of the file (see **OPEN** statement).

LONGMATH values may now be formatted with the **PRINT USING** statement. The format is "*www*[.[*ppp*]][c]". (See "Working With Long Numbers".)

Unformatted printed LONGMATH numbers cannot be read back with **INPUT #** unless they are less than 256 characters long, and contained on one line in the file.

If, instead, you write the LONGMATH value to the file using the **WRITE #** statement, the number will be formatted as a series of quoted strings which can be read with the **INPUT #** statement and assigned to a LONGMATH variable. The format that is written to the file looks like:

"Length=*nnnn*"
"*formatted long number – first part*"
"*formatted long number – second part*"
….
"*formatted long number – last part*"

where the total number of characters in all the strings is nnnn. Each string will be no longer than 78 characters (plus the two quotes), and no longer than the file record length – 2.

When the number is assigned to a variable, it will be rounded to the current floating-pt-length.

You may hand-code a long number in that format if you wish, but it must follow that format exactly, or you could crash the program.

The above format works only for **INPUT #**, not for **INPUT** or **READ**.

Conversion Examples:

Assume *x* and *y* are LONGMATH variables, and *a* and *b* are DOUBLE.

a=b	Normal. DOUBLE assigned without conversion.
x=y	Normal. LONGMATH assigned without conversion.
a=x	*x* is converted, if possible, to DOUBLE and assigned to *a*.
x=a	*a* is converted to LONGMATH and assigned to *x*.
x+a	*a* is converted to LONGMATH and added to *x*. Result is LONGMATH.
a+b+y	*a* and *b* are added without conversion. Intermediate result is DOUBLE, which is converted to LONGMATH and added to *y*. Result is LONGMATH.

a+*x*+*y*	*a* is converted to LONGMATH and added to *x*. Result is LONGMATH, which is added to *y* without conversion.
LONG(*a*/*b*)	*a* and *b* are both DOUBLE, so they are divided, resulting in a DOUBLE, which is converted to a LONGMATH by the **LONG** function.
LONG(*a*)/*b*	The function **LONG**(*a*) results in a LONGMATH. *b* is therefore converted to LONGMATH before doing a LONGMATH divide, resulting in a LONGMATH.
SIN(*a*)	Since *a* is DOUBLE, the short sine function is called, resulting in a DOUBLE.
SIN(*x*)	Since *x* is LONGMATH, the long sine function is called, resulting in a LONGMATH.
SIN(*a*/*x*)	*a* is converted to LONGMATH and then divided by *x*. Since the result is a LONGMATH, the long sine function is called, resulting in a LONGMATH.
x+1e20	The number string is converted to a LONGMATH and added to *x*, resulting in a LONGMATH.
a+1e20	The number string is converted into a DOUBLE and added to *a*, resulting in a DOUBLE.
x+1e2000	The number string is converted to a LONGMATH and added to *x*, resulting in a LONGMATH.
a+1e2000	The number string is too large to convert to DOUBLE, so it is converted to LONGMATH. *a* is then converted to LONGMATH to match it and they are added. Result is a LONGMATH.
b= *a*+1e2000	The same as above, except that the result cannot be converted back into a DOUBLE, so an error occurs.
b=*a*+.12345678901234567 89	Rather than truncate the long (> 16 digits) number, it is converted To a LONGMATH. *a* is then converted to LONGMATH to Match it, and they are added. Result is LONGMATH, which is Converted to a DOUBLE and assigned to *b*.

If you wish to avoid the confusion and issues described above regarding which numbers are long and short, and how and when they are converted, you can choose to work with long numbers only (see "Working With Long Numbers").

Where You Can Use Numbers (besides arithmetic).

DOUBLE and LONGMATH ***variables*** and ***number strings***, or ***expressions*** which evaluate to numbers may be used nearly anywhere a number is called for, provided their values are within the proper range. However, keep in mind that using LONGMATH numbers where the precision is not needed is inefficient and will slow your program down.

file numbers (in I**NPUT #**, **WRITE #**, **PRINT #**, **OPEN** and **CLOSE**)
logical record length (in **OPEN**)
floating point length (in **FLOAT**)
sizes of arrays (in **DIM** *array* (*n*,…) and **LONG** *longmath-array* (*n*,…))
indexes for references to arrays (*a* = *array* (*i*))
logic operations (**IF** *a* = 5.7e25, **WHILE** *x*<10, etc.)
parameters for functions (depending on the function) (*a* = *fn* (*n*, …)
control variables and limits (in **FOR**)
character positions & counts (**LEFT$**, **RIGHT$**, **MID$**, **INSTR**)
--- more

Notes on Precision

In floating-point numbers, you have a defined "floating-point length", which is the maximum number of significant digits in a number, disregarding the decimal point and optional exponent. This defines how the number is stored, and to how many digits a calculation is carried out. Answers are usually rounded to the closest value of the least-significant digit. Therefore, if the floating-point length is 1000, the calculated value of 2/3 would be .6666-------6666**7**, where the 7 is the 1000th digit, and is the result of rounding.

Floating-point numbers are seldom exact, unless the calculation terminated before the floating-point length. So the numbers are said to be "accurate to *n* digits", where *n* is the floating-point length. What does that mean?

If you calculate the square root of a number and save it as a floating-point variable, and then square it, the result will usually not be exactly the same as the original number, and it may not be accurate to *n* digits, due to rounding. So **SQR**(2) * **SQR** (2) may come out to 1.999----9999, but the square root is said to be "accurate to *n* digits" if increasing the last digit by 1 will result in a number whose square goes above 2 (like 2.0000----0001). Similarly, if the square root had been rounded up so that the square was slightly above 2, subtracting one in the last digit would put the result below 2. We can't get any closer than that with *n* digits.

Therefore, the "correct" answer, to *n* digits, is the number that comes closest to the value we want, rounded to the n^{th} digit.

Some functions, like arctangent, lose precision when the argument is very large. For, example, (with a length of 50, and expressing angles in degrees):

TAN (**ATN** (**LONG** (2000000) / 3))=
= **TAN** (89.99991405633073044097643675489837186350707163312**9**)
= 666666.66666666666666666666666666666666666666**596892**.

This appears to have a large error, but when you add 1 to the tangent argument,

TAN (89.99991405633073044097643675489837186350707163313**0**)

= 666666.66666666666666666666666666666666666**7372594**.

The second error is larger, so the calculated arctangent is "correct", since it the closest value which can be expressed in 50 digits.

The "raise to power" function (*x*^*y*) is calculated as **EXP** (**LOG** (*x*) * *y*). This can result in a loss of precision since the result of the **LOG** function is shortened and rounded before using it as an argument to the exponential function. Also, the characteristic of the logarithm occupies digits but does not contribute to the precision of the result, only its magnitude. To compensate for these, the *x*^*y* is calculated at a higher precision and the result rounded back to *n* digits.

You can improve the precision of any function or calculation in the same manner. Simply increase the floating-point length, do the calculation, reset the length, and re-assign the value (which will force rounding to *n* digits. Example:

```
long x
float floatingptlength+8
x= tan (atn (long (2000000) / 3))
float floatingptlength-8
x=x         // forces rounding
print x
```

result: 666666.66666666666666666666666666666666666666666**7**

In a similar manner, you can check the precision of any calculation by doing it at a higher precision and comparing the answers. If the lower precision number matches the first *n* digits of the higher precision number, then those digits can be considered accurate.

Hexadecimal Strings.

The function **HEX$** can create a string representation of a number in hexadecimal. It is, however, useless for LONGMATH numbers or non-integer numbers.

QuickCalc includes a new function: **HEXCONVERT**, which will convert LONGMATH numbers to hexadecimal and back, and it works as follows:

LONGMATH to hex string variable

To convert a LONGMATH number into hex, specify

HEXCONVERT (*num-expr*).

DOUBLE variables and number strings will first be converted to LONGMATH.

The result is a string representing the hexadecimal equivalent of the number. The integer portion of the hex number (plus sign and decimal point) must be less than 255 characters long. The portion to the right of the decimal point will be converted, and carried out to floating_pt_length characters, and will be truncated (not rounded) , if necessary, to fit in the 255-character string. There is **no exponent**.

The resulting string from this form of HEXCONVERT may be assigned to a string variable or used in an expression, i.e.,

"abc" **+ HEXCONVERT** (123.45).

LONGMATH to hex string array.

If the LONGMATH number is too long for a single 255-byte string, it may be converted into a **string array**. The array must have been previously dimensioned, and must contain enough elements to hold the desired length, at 255 bytes per element. Specify the statement:

string-array$ = **HEXCONVERT** (*num-expr*).

num-expr may be a LONGMATH variable or any numeric expression which can be converted into LONGMATH.

This form of **HEXCONVERT** may only be assigned to a string array. It may not be used in an expression, printed, etc. If the resulting string is too long and is assigned to a string variable, it will create an error.

The string array is handled similarly to the way it is used for LONGMATH variables, however the hex strings may not be used in the **LONG** function.

The string array may be printed, written with **WRITE** #, and read back in with **INPUT** #, just like string arrays created from LONGMATH numbers. (The strings are **not** compatible with LONGMATH numbers.)

Hex string to LONGMATH variable.

You can convert hex strings back to decimal with the same function:

longmath-value = **HEXCONVERT** (*string-expr* | *string-array*)

The string expression (or array) must contain no characters other than valid hex digits (0-9 and A-F or a-f), plus an optional sign and only one (optional) decimal point. No other characters, including leading blanks, are permitted.

The result is a LONGMATH value, which can be used in an expression, printed, etc. If it is assigned to a DOUBLE variable, it will be converted, if possible. The precision of the resulting LONGMATH value is rounded to **FLOATINGPTLENGTH** digits.

If a long hex string array was previously written to a file with the **WRITE #** statement and then later read back in to a string array with the **INPUT #** statement, that string array may be converted back to decimal in this manner.

Notes on Hexadecimal Strings.

You **can't do arithmetic** on hex strings. You can convert them to LONGMATH, perform arithmetic or functions on them, and then convert them back to hex strings.

Remember that DOUBLE variables only contain about 16 digits of precision. Although you can convert them to hex, the results may not be what you expect. 1/10 as a DOUBLE is .10000000000000001, and 10^{200} as a DOUBLE is 9.9999999999999997e+199. These differences may seem insignificant and may disappear when rounding, but as LONGMATH values, the differences are noticeable. The hex conversions of the DOUBLE values will be different from the corresponding LONGMATH numbers. If you want exact hex conversions, start with LONGMATH values. For example, use **LONG**(10)**^**200 instead of 10**^**200 to force the calculation to take place using LONGMATH functions.

Fractions which are simple in decimal may result in un-ending or infinitely-repeating strings in hexadecimal. For example, 1/10 is .1 in decimal, but is .199999999…. in hex. Therefore, a number converted to hex and back again may not be exactly the same.

Hex strings are not rounded.

Hex strings are shorter than their corresponding decimal strings. Therefore, a 50-digit decimal fraction will only have about 41 digits of accuracy, even though the string is longer. You may want to increase the floating point length before converting and then truncate the string afterward.

Remember that in floating-point (**FLOAT**) mode, LONGMATH numbers are rounded to the current floating point length, so large integers may not be exact. For large integer calculations, make sure the floating point length (**FLOAT** *nnn*) is longer than the longest integer, or do your calculations in **INTEGER** mode.

String arrays use a **lot** of memory. The default "pool" of memory for string variables is 64K. If you are working with really long numbers, you will probably want to increase that to a least twice the size of all your string arrays combined to avoid running out of string space. It is better to leave the numbers in their LONGMATH form, which doesn't use string space.

The direction of the **HEXCONVERT** function (decimal to hex or hex to decimal) depends on the parameter given. If the parameter is a constant, it is assumed to be hex if it is in quotes, otherwise, it is assumed to be decimal.

If you are in **INTEGER** mode, conversions in either direction will ignore anything to the right of the decimal point and will return an integer. You can also specify **HEXCONVERT** (**FIX** (*num-expr*)) to get an integer result and still remain in floating-point mode.

Negative hex strings are shown with a minus sign. If you want the number to be in 16-complement (where -1 = FFFF…FFFF), specify **HEXCONVERT** (**LONG**(16)**^**n **–** value). Pre-calculate the 16**^**n if you are going to do this a lot.

Note: For accuracy and speed, it is better to calculate large powers of 16 by multiplying, rather than using the power function (16**^**n). For example, calculating 16**^**65536 can be done with the following:

```
LONG b
INTEGER
b=16
b=b*b*b*b*b*b*b*b*b*b*b*b*b*b*b*b  // 16^16
b=b*b*b*b*b*b*b*b*b*b*b*b*b*b*b*b  // 16^256
b=b*b*b*b*b*b*b*b*b*b*b*b*b*b*b*b  // 16^4096
b=b*b*b*b*b*b*b*b*b*b*b*b*b*b*b*b  // 16^65536
```

This gives an exact number 78914 digits long, and although 64 multiplies are required, this is still **a lot faster** than doing a logarithm and exponentiation at that length.

Note: This same calculation can be done in only 16 multiplies (much faster) using the code:

```
b=16
for i = 1 to 16
   b=b*b
next
```

This only works because the exponent (65536) is an exact power of 16.