User-Defined Functions

You can define your own functions within your BASIC program with user-defined functions. The format is:

DEF *function-name* ([*arg1* [, *arg2*]…]) = *expression* | **GOSUB** *stmt-no*

function-name is a valid variable name. This name becomes the name of the function. *function-name* must be unique and may not duplicate the name of any BASIC function, statement, command, constant or operator (see the **PRINTALL RESERVED** statement), and may not be the same as any variable or array name used in your program.

Note: Earlier versions of BASIC used the syntax **DEF FN***name*, where the letters "FN" were considered part of the function name. That syntax is not required in this version, however it will still work.

There are two types of functions:

Single Expression – the function is defined as a single-line expression, exactly like an expression you would assign to a variable (*i.e.*, *var* = *expression*). The expression must evaluate to a number [DOUBLE or LONGMATH], number string, or string, and the function will return that value.

Subroutine – The function is defined as a **GOSUB** *stmt-no* statement. When the user-defined function is executed, the subroutine at *stmt-no* will be called, exactly as when using a **GOSUB**. The subroutine may contain any valid BASIC statements, and must return with a **RETURN** statement. The **RETURN** statement may specify a value to be used as the return value from the user-defined function.

Note: When invoking the subroutine form of a user-defined function, it must be within an expression or assigned to a variable, even if the return value is not used – it cannot be used like a command or statement:

DEF *calculate_results* (*x, y*) = **GOSUB** 1000
calculate_results (2, 4) ← *(invalid)*
a = *calculate_results* (2, 4) ← *(valid)*

The "Subroutine" form has greatly-extended power, since it can make logic decisions, perform input/output, call other subroutines, etc. It can test limits, check for special situations, do table lookup and print error messages. (If the function is the object of a **PRINT** statement (*e.g.*, **PRINT** xxx (*y*)), then you should not print from within the subroutine.)

The value returned must be compatible with the type of variable it will be assigned to or with the way the value is to be used (as a subscript, file number, function argument, etc.)

Argument Substitution

The argument list may be empty, but the parentheses are still required. You may have up to 5 arguments. When the function is called, you may specify no more parameters than appear in the argument list.

The arguments specified in the **DEF** statement temporarily replace the actual variables with the same name, but do not affect those variables. Constants (number strings) supplied as parameters will be converted to numeric variables (DOUBLE or LONGMATH) depending on the size of the number supplied. When you are processing the function, the variable names in the arguments will refer to the arguments rather than the real variables. If you turn **DEBUG** on before and **DEBUG OFF** after you call the function, you can see how the argument substitution is done.

Any argument for which there is no corresponding parameter in the calling statement will be assumed to be zero (or NULL if it is a string), and will ***still*** override the corresponding variables in the expression of subroutine.

You ***cannot*** assign new values to these argument variables. They are parameters, and as such are read-only.

Keep in mind that the type (DOUBLE or LONGMATH) may be different than you expect based on the number supplied in the parameter. If you want to assure that a number is of a certain type, use the **LONG** or **SHORT** functions in the parameter list or within the user function definition:

a = *xxx* (**LONG** (2), *y*)
or **DEF** *xxx* (*x*, *y*) = **SQR** (**LONG**(*x*) + *y*)

Subroutines

A subroutine which is the target of a function definition may also be called from a **GOSUB** statement. Things to consider when doing this are:

There are no over-riding variables. Variables which would have been over-ridden for a function are now normal variables which may be assigned values. This can lead to confusion if the subroutine is used both ways.

The return value will be calculated but ignored.

The same subroutine could be the object of two different function definitions, which override different arguments. This could get confusing.

Nesting.

Function definitions can be nested. Expression forms of user-defined functions may not recursively call each other (*e.g.*, **DEF** *aaa*(*x*) = *bbb*(*x*): **DEF** *bbb*(*x*) = *aaa*(*x*)); to do so could crash the program. Subroutine forms may do recursive calls (but ***be careful***!) See examples below.

To avoid crashing the program, nested user-defined functions are **limited to 200 deep**, after which the BASIC program will terminate with an error message.

If a function definition expression or subroutine references another user-defined function, the inner function's arguments are set by the outer function with values that may have been passed to the outer function as parameters. When referencing variables, the current values of the inner function's arguments are searched first. If not found there, the outer function's arguments are searched, and then the variables in the main program.

Thus, a subroutine or function may be referencing variables from several sources. It is best to name your variables and arguments carefully to avoid confusion. Variables considered "local" by the subroutine should have names not used elsewhere in the program and not used as parameters. You should not dimension arrays or define long numbers from within the subroutine.

LONGMATH Numbers in User-Defined Functions

Within the *expression* or subroutine, LONGMATH numbers may be used along with DOUBLE numbers, just as with regular expressions. The type of numbers used as parameters will affect the type of computation and the type of output. For example, if you had

LONG *x*
DEF *xx* (*a*, *b*) = **SQR** (*a* / *b*)
y = *xx* (*c*, *d*) would do a DOUBLE divide and square root,
y = *xx* (*x*, *d*) would do a LONGMATH divide and square root,
y = *xx* (123, *x*) would do a LONGMATH divide and square root,
y = *xx* (**SHORT** (*x*), *c*) would do a DOUBLE divide and square root.

See "Number Representation, Assignment and Conversion", section "Conversion Examples" for more information and examples.

Trigonometric Functions

Similarly, trigonometric functions will behave differently depending on whether **DEGREES** or **RADIANS** (default) was specified most recently. Therefore, you don't need to define separate functions for working with angles in degrees *vs.* radians. (If you need to check which mode you are in, use the **CHECKDEGREES** function).

Examples:

```
// nested definition:

DEF tangent (xsin) = xsin / (SQR (1-xsin*xsin))     // |xsin| < 1
DEF arcsin (xsin) = ATN (tangent (xsin))     // computes arcsine
angle = arcsin (sine)

// sum of all odd numbers up to n (recursive):

DEF oddsum (n) = GOSUB 100
x = 0  //    x must start at 0
PRINT oddsum (29)
END
100 IF n % 2 = 0 THEN y = oddsum (n-1): RETURN x
x = x + n
IF n >= 3 THEN y = oddsum (n - 2)
RETURN x
```

Uses for Hyperbolic Trig Functions, etc.

You can create user-defined function definitions for functions not provided as part of BASIC.

```
Maximum:        DEF max (a, b) = (a > b) * a + (b >= a) * b
Hypoteneuse:    DEF hypoteneuse (a, b) = SQR (a*a + b*b)

Secant:         DEF sec (x) = 1 / COS (x)                    // | x|  < 1
Cosecant:       DEF csc (x) = 1 / SIN (x)                    // x  ≠ 0

ArcSine:
        DEF arcsin (x) = ATN     (x / SQR (1 - x* x)         // | x|  < 1 *
ArcCosine:
        DEF arccos (x) = ATN     (SQR (1 - x*x) / x)         // |x| ≤  1,  x  ≠ 0
ArcSecant:
        DEF arcsec (x) = arccos (1 / x)                      //  |x| ≥  1,  x  ≠ 0
ArcCosecant:
        DEF arccsc (x) = arcsin (1 / x)                      // |x| > 1,    x  ≠ 0
```

* The arcsine is actually defined for all $|x| \leq 1$, however this function definition blows up at ±1. This is where the subroutine form of user-defined function can help.

```
DEF arcsin (x) = GOSUB 1000
1000 IF ABS (x) = 1 THEN RETURN SGN (x) * ACOT (x-x)
RETURN  ATN  (x / SQR (1 - x* x))
```

(*x-x* creates a long or short zero, based on the type of *x*, and **ACOT** (0) is π/2 or 90˚, depending on the degrees/radians setting.)

The above "subroutine" function works for all values of $|x| \leq 1$. It is obvious that the same technique would work for the arccosine, and any other function whose definition depends on the range of the argument.

Hyperbolic Sine: **DEF** *sinh* (*x*) = (**EXP** (*x*) – **EXP** (-x)) / 2
Hyperbolic Cosine: **DEF** *cosh* (*x*) = (**EXP** (*x*) + **EXP** (-x)) / 2
Hyperbolic Tangent: **DEF** *tanh* (*x*) = *sinh* (*x*) / *cosh* (*x*)
Hyperbolic Cotangent: **DEF** *coth* (*x*) = *cosh* (*x*) / *sinh* (*x*)
Hyperbolic Secant: **DEF** *sech* (*x*) = **1** / *cosh* (*x*)
Hyperbolic Cosecant: **DEF** *csch* (*x*) = **1** / *sinh* (*x*)

ArcHyperbolic Sine: **DEF** *arcsinh* (*x*) = **LOG** (*x* + **SQR** (*x* * *x* + 1))
ArcHyperbolic Cosine:
DEF *arcsinh* (*x*) = **LOG** (*x* + **SQR** (*x* * *x* - 1)) : // $x \geq 1$
ArcHyperbolic Tangent:
DEF *arctanh* (*x*) = (1/2) * **LOG** ((1 + *x*) / (1 - *x*)) : // $x < 1$
ArcHyperbolic Secant:
DEF *arcsech* (*x*) = **LOG** ((1 + **SQR** (1 – *x* * *x*) / *x*) : // $0 < x \leq 1$
ArcHyperbolic Cosecant:
DEF *arccsch* (*x*) = **LOG** (1/*x* + **SQR** (1 + *x* * *x*) / **ABS** (*x*)): // $x > 0$
ArcHyperbolic Cotangent:
DEF *arccoth* (*x*) = (1/2) * **LOG** ((*x*+1) / (*x*-1)) : // $|x| > 1$