PSIM

TUTORIAL

How to Use Script Functions

February 2021

The Script functions that used in PSIM's **Script Tool** can perform calculations, run simulation, and plot graphs. The **Script Tool** is in the **Script** menu.

Getting Started

Clicking on the **Script** menu, then, selecting **Script Tool** from the pulldown list, will open dialog of the **Script Editor**. The editor contains the following menus:

- File: Create a new, Open, Save or Close a script file
- Edit: Find or Replace texts, and some display options of the editor dialog
- Run: Check syntax and Run script,
- Help: Online help of the Script Tool and its functions.

A script file is a text file. It supports everything that a PSIM element **Parameter File** supports. In addition, it supports the plot and simulate functions, and much more.

For the basics of the data types, statement format and syntax, please refer to the online help of the Parameter Tool and Script Tool.

The script tool supports the following function categories:

- Computational functions
- Operators
- Control functions
- Arrays
- Strings
- Complex numbers
- Dictionary
- Matrix
- File functions
- Graph function
- Simulation function
- Other

This tutorial describes how each of the functions is defined and used.

Each script function has its own help page. To go to the help page of a function, highlight the function name in the Script Editor, and press the F1 key or click Help menu.

For example, in a script file shown here:

```
// Plot amplitude and phase angle
Plot({"Amplitude", "xlog"}, arrGraphs[0], arrGraphs[1]);
Plot({"Phase", "xlog"}, arrGraphs[0], arrGraphs[2]);
```

Highlighting the function **Plot**, then clicking the **Help** menu, will take you to the online help page of the **Plot** function.

Below is an example of a script. It shows the current loop design of a buck converter. For more details, refer to the document "Tutorial – Control Loop Design Using Script Functions.pdf".

```
// Buck converter - current loop design
PI = 3.14159;
// Parameters
Vin = 250;
L1 = 200u;
C1 = 245u;
R1 = 0.6;
Ksen_i = 1/165; // current sensor gain
Ksen_v = 1/100; // voltage sensor gain

// PWM gain: Kpwm = Vin/Vcarrier, with Vcarrier=1
Kpwm = Vin;
fsw = 20k;

// Current PI controller
Ki_pi = 1.24292;
Ti_pi = 142.139u;

// Plant: Hi(s) = iL / Vos
Hi = formula(1/(s*L1 + R1/(s*R1*C1+1))); // define Hi(s) as a formula

// Current PI: Gi = Ki * (1+sTi) / (sTi)
Gi = formula(Ki_pi*(1+s*Ti_pi)/(s*Ti_pi)) // define Gi(s) as a formula

// Current loop transfer function Ti_loop
Ti_loop = Hi*Gi*Ksen_i*Kpwm // perform formula calculation

// Bode plot
fmin = 100;
fmax = 50e3;
Freq_Hz = ArrayLog(fmin, fmax); //define frequency array (Hz) from fmin to fmax, in log
Freq_rad = Freq_Hz*2*PI; // define frequency array (rad/sec)
Print(Freq_Hz);                 // Display the content of the Freq_Hz array

s = Complex(0, Freq_rad); // define Laplace operator

BodePlot("Current Loop", Freq_Hz, Hi, Gi, Ti_loop); // generate Bode plots for Hi, Gi, and Ti_loop
```

In the script, a double forward slash represents comments.

It is strongly recommended that each line of the script statement ends with a semicolon ";". This makes it much easier to parse the script and less likely to have a mistake.

Data Types

Supported data types in the script are:

- float,
- integer,
- complex,
- string, and
- array

Note that data type declaration is not needed. When a variable is assigned, its data type is determined automatically. The same variable can change its data type through another assignment. For example, below is an example of data type assignment in the script:

```
// This script shows how variable data types are assigned.
i1 = 5;              // i1 is an integer
r1 = 4.6;                    // r1 is a float
c1 = complex(1.1,2.2); // c1 is a complex number
s1 = "test1";                // s1 is a string
s2 = s1+"test2";             // s2 is also a string
s1 = c1;             // s1 is now a complex number
a1 = {1, 2, 3, 5, 7};        // a1 is an integer array. a1 = {1,2,3,5,7}
a2 = array(1, 5, 1);         // a2 is an integer array. a2 = {1,2,3,4,5}
a3 = array(complex(2,-1), complex(10,-5), complex(2,-1)); // a3 is a complex array.
                             // a3 = {2-j1, 4-j2, 6-j3, 8-j4, 10-j5}
Months = {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec} // Months is a string array
```

Also, script function names are not case sensitive. But variable names are case sensitive.

Supported script functions are described in the sections below.

Script Run Display

By default, variable values at each step of the calculation will be displayed in display window. To clear the display window, from the Edit menu, choose:

Clear message window
Clear message before each run

Displaying variable values at each step will slow down the script run. A function **ScriptOption** is provided to disable or enable the display.

Computational Functions

The following computational functions are supported:

abs(x)	Absolute value
acos(x)	Arccosine, in radian
acosd(x)	Arccosine, in degree

arccos(x)	Arccosine, in radian
asin(x)	Arcsine, in radian
asind(x)	Arcsine, in degree
arcsin(x)	Arcsine, in radian
atan(x)	Arctangent, in radian
atand(x)	Arctangent, in degree
arctan(x)	Arctangent, in radian
atan2(y,x)	Arctangent with x and y defined, in radian
atan2d(y,x)	Arctangent with x and y defined, in degree
ceil(x)	Function that returns the integer larger than x
cos(x)	Cosine (x in rad.)
cosd(x)	Cosine (x in degree)
cosh(x)	Hyperbolic cosine
eval	Evaluate a formula
exp(x)	Exponential of x (with base e, that is, e^x)
FFT	FFT analysis
floor(x)	Function that returns the integer smaller than x
formula	Define a formula as a function of variables
hypot(x1,x2,x3,…)	Square root of x1 squared plus x2 squared, plus x3 squared, etc., i.e. sqrt(x1^2 + x2^2 + x3^2 + …)
hypot(x_array)	The input is an array, and it returns the square root of the sum of the array cells squared, i.e. sqrt(x_array[0]^2 + x_array[1]^2 + x_array[2]^2 + …)
ln(x)	Natural logarithm of x (with base e)
log(x)	Natural logarithm of x (with base e)
log10(x)	Common logarithm of x (with base 10)
max(x1,x2,x3…)	Maximum value of x1, x2, x3, etc. (no limit on the number of inputs)
max(x_array)	The input is an array. It returns the maximum value of the array cells
min(x1,x2,x3,…)	Minimum value of x1, x2, x3, etc. (no limit on the number of inputs)
min(x_array)	The input is an array. It returns the minimum value of the array cells
mod(x,y)	Modulo function that returns the reminder of x/y. This is the same as the % operator. For example, mod(5,2) = 1.
pow(x,y)	x to the power of y
pwr(x,y)	Absolute value of x to the power of y, i.e. abs(x)^y
sign(x)	Sign function that returns 1 if x > 0; -1 if x < 0; and 0 if x = 0
sin(x)	Sine (x in rad.)
sinh(x)	Hyperbolic sine

sqr(x)	Square of x, that is, x^2
sqrt(x)	Square root
tan(x)	Tangent (x in rad.)
tanh(x)	Hyperbolic tangent

FFT Analysis

The FFT function performs FFT analysis. There are two ways to use the FFT function, as shown below.

DoFFT(arr, Xmin, Xmax)

Perform FFT analysis on the data stored in the object "arr" for the range from Xmin to Xmax

DoFFT(array_y, array_x, Xmin, Xmax)

Perform FFT analysis on the data stored in the array array_y with the time stored in array_x for the range from Xmin to Xmax

For DoFFT(arr, Xmin, Xmax), arr is an object that contains the SIMVIEW data from simulation. It is assumed that the first column of the data is time.

For DoFFT(array_y, array_x, Xmin, Xmax), array_y is an array that contains the data for FFT analysys, and array_x contains the time data.

Xmin and Xmax define the start time and end time of the data used for FFT analysis. They are optional. If they are not defined, the entire data is used.

The data range, defined as Xmax-Xmin, must be an integer number of the fundamental cycle. For example, if the fundamental frequency is 1kHz, the data range should be 1m, or 2m, or 3m, etc.

Below is an example how FFT is called if the data comes from simulation:

```
file_smv = GetLocal("PARAMPATH") + "buck converter.smv";

// Read the SIMVIEW file into the object arr.
arr = GraphRead(file_smv);

// Perform FFT on the curve Vos. It is assumed that the first column of the data is time. Only the
data from 0.4ms to 0.5ms is used for FFT.
fft_result = DoFFT(arr["Vos"], 0.4m, 0.5m);
```

Below is an example if the time and data arrays are defined separately.

```
// Define the time array and data array
array_x = Array(0, 6*pi, pi/1000);
array_y = sin(array_x);

// Perform FFT on array_y
fft_result = FFT(array_y, array_x, 2*pi, 4*pi);
plot(fft_result);
```

Formula Functions

Formula functions define a formula and evaluate a formula. The functions are described below.

Formula(math expression)

This function defines a formula as a function of variables. Values of the variables have not been defined. For example,

```
f1 = Formula(a+b*c);     // f1=a+b*c where a, b, and c are variables with unknown values
```

Eval(formula name, variable1=value1, variable2=value2, …)

This function evaluates the value of a formula based on variable values. For example,

```
f1 = Formula(a+b*c);     // f1 = a+b*c where a, b, and c are variables
f1_value = Eval(f1, a=1, b=2, c=3);          // evaluate f1 with a, b, and c at the specified values
```

One can perform addition, subtraction, multiplication, and division operations on formulas.

Below is an example of a script using formula functions.

```
H = formula(R / ((s * R * C) + 1));// define the transfer function H(s)
// where s, R, and C are variables
G = formula(kpi *((s*Tpi)+1) / (s*Tpi));    // define the transfer function G(s)
T = G*H;          // perform formula multiplication G(s)*H(s)
Tcl = T/(1+T);    // perform formula math operations
```

Operators

The following mathematical operators are supported:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo operator that returns the remainder after division (e.g. 5 % 2 = 1)
^	To the power of (i.e. x^y returns x to the power of y)
**	To the power of (i.e. x**y returns x to the power of y)
=	Equal
==	Conditional equal
!=	Not equal
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
!	Not operator
&&	And

| | Or

Control and Error/Warning Functions

The following control and error/warning functions are supported:

if (...)
{...}
else if (...)
{...}
else
{...}
	Conditional if statement

iif (condition, value1, value2)	Inline if statement (note it is “iif”, not “if”)
While (...) {...}	While loop
Error (“text %f, %f”, var1, var2)	Error statement. Up to 5 variables are supported.
Warning (“text %f, %f”, var1, var2)	Warning statement. Up to 5 variables are supported.
Return (var1)	Return the value var1, and end the execution.

Example:

Below is an example of the if and while statements:

```
if (k1 > 10)
{
	a1 = 10;
}
else
{
	a1 = 5;
}
iflag = 1;
b1 = 0;
while (iflag == 1)
{
	b1 = b1 + 0.1;
	if (b1 > 10)
		iflag = 0;
}
```

Array

An array is defined using a pair of curly brackets, with items separately by comma. Below are some examples:

```
Months = {Jan, Feb , Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
a = {1, 2, 3, 5, 7, 11};
b = {10.5, 8.3, 33.59};
```

The value of a cell in an array is accessed by using the square bracket [ ]. For example, in the examples above, Months[0] is "Jan", a[1] is 2, and b[1] is 8.3.

A cell value can be changed in the same way. For example, a[1] = 5;

A multi-dimensional array can be defined by nesting brackets inside each other. For example:

```
M = {{1,2}, {3,4}, {5,6}};
```

where M[0][1] is 2. It can be changed to 50 by the statement: M[0][1] = 50;

Some examples are shown below.

```
a = {1, 2, 3, 4, 5}; // original array
a[1] = 10; // This change a[1] to 10 also.
c = a; // A new array is created. Changes to c does not affect a.
c[2] = 50; // a[2] is not affected.
```

Array Arithmetic

When two arrays of different sizes are used in an arithmetic operation, the larger array will be shrunk to the size of smaller array. The resulting array's size will be equal to the size of the smaller array.

Below is an example:

```
arr1 = {10, 100, 1000, 10000};
arr2 = { - 1, - 2};
arrA = arr1 + arr2;   // result is {9,98}
arrB = arr1 - arr2;   // result is {11,102}
arrC = arr1 * arr2;   // result is {-10,-200}
arrD = arr1 / arr2;   // result is {-10,-50}
arrE = arr2 / arr1;   // result is {-0.1,-0.02}
arrF = log10(arr1);   // result is {1,2,3,4}
arrG = 6 * arr2;  // result is {-6,-12}
arrH = 6 / arr2;  // result is {-6,-3}
```

Supported array functions are described below.

AddToArray(Var1, value1, ….)

This function adds additional cells to 'Var1', and returns the result back to 'Var1'. Note that the variable 'Var1' must be an array.

In the example below, three values are added to the array a:

```
a = {1,2,3,4}
AddToArray(a, 5,6,7);   // a  is  modified to:  {1,2,3,4,5,6,7}
```

Empty brackets [] also performs the same task.

```
a = {1,2,3,4}
a[] = 5;
a[] = 6;
a[] = 7;
// a is modified to: {1,2,3,4,5,6,7}
```

AppendArray(array1, array2, array3, ….)

This function returns an array that is the result of appending all arrays in the parameter list, in the same order as they appear. All arrays in the parameter list must be of the same type.

This function also applies to graphs.

In the example below, the array arr2 is appended to the array arr1:

```
arr1 = {1,2,3,4};
arr2 = {10, 11};
arr3 = AppendArray(arr1,arr2);   // arr3  is:  {1,2,3,4,10,11}
```

Array(size)
Array(size, InitialValue)
Array(FirstValue, LastValue, increment)

The Array function defines a one-dimensional array of size ‘size’. It can initialize all cell values to ‘InitialValue’. It is also possible to initialize the array to a set of numbers starting at ‘FirstValue’ and increasing by ‘increment’ at each cell until it reaches ‘LastValue’.

For example, the following example defines an array of 10 cells and initializes them with a while loop. The result array is: x1 = {15,17,19,21,23,25,27,29,31,33}.

```
x1 = Array(10);
i = 0;
while (i < 10)
{
   x1[i] = 15 + (2 * i);
   i++;
}
```

In another example, x2 = Array(15, 33, 2) defines an array of 10 cells and initializes them with the Array function. The result array is: x2 = {15,17,19,21,23,25,27,29,31,33}

This function also works for decreasing numbers. For example, x = Array(9, 3, 1.5) defines the array x = {9, 7.5, 6, 4.5, 3}.

The following example defines an array of 10 cells and initializes it to “Row “. It then adds numbers in a while loop. The result array is: L = {Row 1,Row 2,Row 3,Row 4,Row 5,Row 6,Row 7,Row 8,Row 9,Row 10}.

```
L = Array(10, "Row ");
i = 0;
while (i < 10)
{
   L[i] = L[i] + string(i+1);
   i++;
}
```

ArrayLog(FirstValue, LastValue)

This function defines a one-dimensional array and initializes it to set of numbers starting at ‘FirstValue’ and ending at ‘LastValue’. Numbers are increased in a manner suitable for use with exponential calculations.

For example,

```
X = ArrayLog(8,300); // X = {8,9,10,20,30,40,50,60,70,80,90,100,200,300}
Y = ArrayLog(10000, 500); // Y = {10000,9000,8000,7000,6000,5000,4000,3000,2000,1000,900,800,700,600,500}
```

SizeOf(Var1)

If ‘Var1’ is a one-dimensional array, this function returns the size of the array. If ‘Var1’ is a string, this function returns the size of the string. If ‘Var1’ is a curve, it returns the numbers of rows.

When the input is text string, sizeof returns the length of string. When the input is an array, sizeof returns the number of cells in the array. When the input is a graph, sizeof returns the number of rows in the graph.

Following is a sample code:

```
s1 = "1.56";
len = sizeof(s1); // len is 3

arr = {0,1,2,3,4,5,6,7};
a = sizeof(arr); // a is 8
```

Copy(arr)
Copy(arr, StartIndex, Length)

The Copy function returns a full or partial copy of an array.

The parameters are:

arr :	Input array.
StartIndex:	Zero based index of location of array cell to start the copy from.
Length:	Length of returned array

The function Copy(arr) returns a full copy of an array. The function Copy(arr, StartIndex, Length) returns a partial copy of the array starting from 'StartIndex' and continuing for the 'Length' number of cells.

If 'StartIndex' is -1, then it copies the last 'Length' cells. If 'Length' is -1, then it copies all cells after 'StartIndex' (including 'StartIndex') to the end of the input array.

String

A string is defined by characters between two quote " signs, for example, "This is a string".

Strings can be placed in variables, as shown below.

```
Var1 = "Apple";
Var2 = "Orange";
```

The plus sign + is used for combining strings. Other arithmetic signs - * / are not valid when dealing with strings. For example,

```
Var3 = Var1 + Var2; //  Content of Var3 is:   "AppleOrange"
```

A special object, _CRLF, is defined as carriage return line feed, and can be used in the string operations.

Functions related to strings are described below.

Find(InputString, StringToFind, StartSearchIndex(optional))
Findreverse(InputString, StringToFind, StartSearchIndex(optional))

The Find function looks for a character or a sub-string in the Input string. It starts the search forward from the 'Start search index' value if present. The function returns the zero based index of the location or -1 if it does not find anything.

The Fundreverse function works in the same way as the Fund function, except that it searches in the reverse direction.

Function parameters are:

Input string:	Input string
String to find:	String to find
Start search index:	Zero based index of location of characters to start the search from.

The function returns the zero based index of the location of the first character of the matched substring, or -1 if it does not find anything. Examples are:

```
Var3 = "AppleOrange"
iA = Find(Var3, "A");                 //  iA  is  0  in "AppleOrange",  'A' is the first character
ia = Find(Var3, "a");                 //  ia is  7   pointing to 'a' in Orange
ie1 = Find(Var3, "e");     //  ie1  is 4
ie2 = Find(Var3, "e", ie1+1); // ie2 is 10. Find function starts the search at the character after the
                                      first 'e'
```

Makelower(str)

Convert all characters in a string to lower case.

Makeupper(str)

Convert all characters in a string to upper case.

Replace(Input_string, part_to_replace, part_to_replace_with, part_to_replace, part_to_replace with, ...)

This function replaces parts of the strings with new strings.

This function allows multiple replacement actions by adding unlimited pairs of 'string to replace' and 'string to replace with'. For example, to convert "(2.0, 3.0)(4.5, 8.3)" to "2.0 3.0 4.5 8.3 ":

```
str1 = "(2.0, 3.0)(4.5, 8.3)"  // Remove all input parentheses by replacing them with empty strings
str2 = Replace(str1, "(", "");   //        Replace "(" with nothing
str2 = Replace(str2, ")", " ");  //        Replace ")" with space
str2 = Replace(str2, ",", " ");  //        Replace "," with space
str2 = Replace(str2, "  ", " "); //        Replace two spaces with one space
str2 = Replace(str2, " ", "");  //        Replace all spaces with no space
str3 = Replace(str1, "(", "", ")", " ", ",", " ", "  ", " "); // Alternatively, do all above operations
                                                  // at once. str1 is unchanged.
```

Sizeof(Var1)

This function returns the length of string or size of array or graph

If 'Var1' is a one - dimensional array, this function returns the size of the array. If 'Var1' is a string, this function returns the size of the string. If 'Var1' is a curve, it returns the numbers of rows.

When the input is text string, sizeof returns the length of string. When the input is an array, sizeof returns the number of cells in the array. When the input is a graph, sizeof returns the number of rows in the graph.

Following is a sample code:

```
s1 = "1.56";
len = sizeof(s1); // len is 3

arr = {0,1,2,3,4,5,6,7};
a = sizeof(arr); // a is 8
```

Split(Input_string, Separator_characters)

This function splits a string into pieces using the separator characters. When any of the characters in the string is encountered, the input string is split into another piece. The return value is an array of strings.

For example,

```
Split("22,33,44,55,66", ",");   //return value is array of strings: {"22"," 33"," 44"," 55"," 66"}
```

Split2(Input_string, Start_string, End_string)
Split2(Input_string, Separator_string)

This function splits a string into pieces. The return value is an array of strings. Split2 expects a full string ("Begin") as separator while Split uses a list of characters (" \t\r\n") as separators. Any of the characters in the list would be enough to split the string.

For example,

```
Split2("(22, 33), (44, 55), (66 , 88)", "(", ")");   //returns:  {"22, 33"," 44, 55"," 66 , 88"}
```

String(val)

The function converts a numerical value to a string. For example,

```
a = String(12.7m);                  //  a is "0.0127"
b = String( 15 + 6 );               //  b  is  "21"
arr = {1, 5, 9};
c = String(arr);            //  c is "{1,5,9}"
```

Trim(str)

Trims the leading and the trailing characters from the string.

Trimleft(str)

Trims the leading characters from the string.

Trimright(str)

Trims the trailing characters from the string.

Value(str)

The function converts a string value to a number.

When reading numbers from a text file, it is usually read as strings. These strings must be converted to numbers using the 'Value()' function before they are used in arithmetic operations. For example,

```
s1 = "1.56";
a = string(s1); // a is 1.56
```

SubStr(InputString, StartIndex, Length)

The function returns part of the input string.

Function parameters are:

Input string	Input string
Start index:	Zero based index of location of characters in input string
Length:	Length of returned string

The function returns a string that contains a copy of the specified range of characters starting from 'StartIndex' and continuing 'Length' number of characters.

If 'StartIndex' is -1 then SubStr returns the last 'Length' characters. If 'Length' is -1 then SubStr returns all characters after 'StartIndex' (including 'StartIndex'). Examples are:

```
Var3 = "AppleOrange";
s1 = SubStr(Var3, 3, 5);    // returns "leOra"
s2 = SubStr(Var3, -1, 3);   // returns  last three characters "nge"
s3 = SubStr(Var3, 3, -1);   // returns  all characters after (and including) the  3rd index "leOrange"
```

Complex Numbers

The following complex number functions are provided:

Abs(c1)

This function returns the magnitude of a complex number defined as sqrt(a^2 + b^2)

Angle(c1)

This function returns the angle of a complex number in rad, and it is defined as atan(b, a).

Complex(a, b)

This function returns a complex number. 'a' is the real part, 'b' is the imaginary part. To create an array of complex numbers, each parameter can be an array. For example,

```
c1 = complex(5, 3);  // c1 is:    5 + 3j
arr1 = complex(Array(0,6, 2), Array(-20, -14, 2));  // arr1 is:  {  0–20j, 2-18j, 4-16j, 6-14j }
w1 = complex(5, {1, -5, -7, 22});  // arr1 is:  {  5+j, 5-5j, 5-7j, 5+22j }
```

Imag(c1)

This function returns the imaginary part of a complex number. The following is a sample code:

```
    c1 = complex(5, 3); // c1 is: 5 + 3j
    i1 = Imag(c1); // i1 is: 3
```

Polar(mag, ang)

This function returns a complex number. 'mag' is the magnitude and 'ang' is the angle in rad. To create an array of complex numbers, each parameter can be an array. For example,

```
c1 = Polar(10, 1.4);  // c1 is:    9.85 + 1.7j
arr1 = Polar({10, 20, 30}, {0, 0.7, 1.4});
```

Real(c1)

This function returns the real part of a complex number.

Dictionary

Dictionary data structure, also called associative array is a collection of key-value pairs. Each key is a unique text that only appears once in each dictionary object and maps the key to its associated value. Value is any data type. Value could be a number, string, array, graph, Matrix, or another dictionary object. Key is a case sensitive text, generally less than 50 characters.

The values in a Dictionary are accessed by using the square bracket []. For example, in Dictionary object 'dict' to set the value 5 for the key "R1" we write: dict["R1"] = 5; and to retrieve that value and place it in another variable called 'b' we write: b = dict["R1"]

Following sample code defines a Dictionary object and adds key/value pairs to it:

```
dict1 = Dictionary();
dict1["alpha"] = {1, 2, 3, 5, 7, 11};
dict1["beta"] = 19.2m;
dict1["omega"] = "This text is a value placed in a dictionary";
```

Assign a different value to beta. It will replace the old value

```
dict1["beta"] = 4;
```

Read values from Dictionary:

```
 b = dict1["beta"];
arr = dict1["alpha"];
```

Simulate function accepts Dictionary object as input. This is another way to provide the circuit with parameters. If circuit needs 'R1'and 'R2':

```
d = Dictionary();
d["R1"] = 1k;
d["R2"] = 12.5k;
Simulate(SchematicFilePath, SimviewFilePath, d, ReturnGraph);
```

LoadScript function, loads an external script or parameter file and returns the result in a Dictionary object.

When **FileWrite(FilePath, dict1)** is used to write a Dictionary object to file, it generates a PSIM compatible parameter file.

To iterate through the list of Name/Value pairs, use the function DictionaryToArray to convert the Dictionary to a two dimensional array and use a while loop on the array.

```
arr1 = DictionaryToArray(dict1);
i = 0;
len = sizeof(arr1);
while(i < len)
{
   name = arr1[i, 0];
   val = arr1[i, 1];
   //...;
```

```
    i++;
}
```

DictionaryToArray

DictionaryToArray converts a Dictionary object to a two dimentional array or two separate one dimensional arrays.

```
dict1 = Dictionary();
dict1["R1"] = 1k;
dict1["R2"] = 12.5k;
DictionaryToArray(dict1);
```

When DictionaryToArray(dict1) is used on dict1, it produces the following 2D array:

```
{{"R1", 1000},{"R2", 12500}}
```

Following sample code iterates through the array to find all name/value pairs:

```
arr1 = DictionaryToArray(dict1);
i = 0;
len = sizeof(arr1);
while(i < len)
{
   name = arr1[i, 0];
   val = arr1[i, 1];
   //...;
   i++;
}
```

Table

ReadTable(FilePath, type)

This function reads a lookup table and returns an object containing that table.

GetTableValue(table, x)

This function takes a lookup table (returned by ReadTable) and a 'x' value and returns the result.

GetTableValue(table, x, y)

This function takes a 2D lookup table (returned by ReadTable) and 'x' and 'y' values and returns the result.

GetTableValue(table, x, y ,z)

This function takes a 3D lookup table (returned by ReadTable) and 'x', 'y', 'z' values and returns the result.

FilePath is the full file path of the table.

Valid values for type:

XY	Lookup table.
2DInterpolation	2-dimensional lookup table (with interpolation)
2DInteger	2-dimensional lookup table (with integer input)
3DInterpolation	3-dimensional lookup table with interpolation

Matrix

Matrix(row, column);
Matrix(row, column, 2D_array);

This function returns a matrix object.

Create a matrix object by specifying the number of rows, columns and an optional two-dimensional array for initialization.

Following is a 2x3 Matrix:

```
m1 = Matrix(2, 3, { {1,2,3}, {40,50,60} } );
```

Same Matrix could also be created and initialized as Follows:

```
m1 = Matrix(2, 3);
m1[0, 0] = 1;
m1[0, 1] = 2;
m1[0, 2] = 3;
m1[1, 0] = 40;
m1[1, 1] = 50;
m1[1, 2] = 60;
```

Matrix multiplication and addition are possible using the following syntax:

```
m3 = m1 * m2;
m3 = m1 + m2;
```

Following Matrix functions are provided:

Transpose(m1)

Returns the transpose of matrix m1

Following code create a 3x3 matrix and initialize it to a 2D array and then takes its transpose:

```
m = Matrix(3, 3, { {1,2,3}, {4,5,6} , {-1, 3, 9} } );
mT = Transpose(m);
```

Eigen(m1)

Returns the eigenvalues of matrix m1.

Following code create a 3x3 matrix and initialize it to a 2D array and then takes its eigenvalues.

```
m = Matrix(3, 3, { {1,2,3}, {4,5,6} , {-1, 3, 9} } );
ei = Eigen(m);
```

Inverse(m1)

Returns the inversion of matrix m1

Following code create a 3x3 matrix and initialize it to a 2D array and then takes its inverse:

```
m = Matrix(3, 3, { {1,2,3}, {4,5,6} , {-1, 3, 9} } );
mINV = Inverse(m);
```

Determinant(m1)

Returns the Determinant of matrix m1

Following code create a 3x3 matrix and initialize it to a 2D array and then takes its Determinant:

```
m = Matrix(3, 3, { {1,2,3}, {4,5,6} , {-1, 3, 9} } );
d1 = Determinant(m);
```

File Functions

The following file functions are provided:

FileAppend(FilePath, Object, "A"(optional))

The function appends an object to the end of an existing text file. If the file does not exist, it creates a new file. If the folder does not exist, it creates the folder.

If Object is a string or a number, it is added to the text file. If Object is an array, each cell is written as a separate line. If Object is a graph, first line is the graph name followed by one number per line.

If the file already exists, existing format is used. Only when the file is being created, if the optional "A" character is used, file is written in ANSI format otherwise, default Unicode format is used.

FileRead(FilePath)

The function reads a text file and place the content in a string. For example,

```
str = FileRead("C:\Powersim\MyText.txt");   //  str is a string containing the file content
```

This function is not meant to read SIMVIEW graph files. It is used to read basic text files, and string functions can be used to parse the information. For example, assume that "MyText.txt" contains the following content:

```
// This is a parameter file
R1 = 1.2;
R2 = 2.3;
```

The following code will read the file, and find the value of R1:

```
str1 = FileRead("C:\Powersim\MyText.txt");  // str1 contains strings of the entire file
index1 = Find(str1, "R1");    // Find the index of string “R1”
index2 = Find(str1, "=", index1);   // Find the equal sign after index1
index3 = Find(str1, ";", index2);   // Find the semicolon after index2
R1_value_str = SubStr(str1, index2+1, index3-index2-1);  // Find the value of R1 in string
R1_value = Value(R1_value_str);  // Convert the string to a number
```

FileReadLines(FilePath)

The function reads a text file and place the content in an array of strings. Each array cell contains one line of text. For example,

```
arr = FileReadLines(“C:\Powersim\MyText.txt”);   //  arr is an array of strings.
```

This function is not meant to read SIMVIEW graph files. It is used to read basic text files, and string functions can be used to parse the information. For example, assume that “MyText.txt” contains the following content:

```
// This is a parameter file
R1 = 1.2;
R2 = 2.3;
```

The following code will read the file, and find the values of R1 and R2:

```
str1 = FileRead("C:\test\MyText.txt");  // str1 contains strings of the entire file
index1 = Find(str1, "R1");    // Find the index of string "R1"
index2 = Find(str1, "=", index1);   // Find the equal sign after index1
index3 = Find(str1, ";", index2);   // Find the semicolon after index2
R1_value_str = SubStr(str1, index2+1, index3 - index2 - 1);  // Find the value of R1 in string
R1_value = Value(R1_value_str);  // Convert the string to a number
```

FileWrite(FilePath, Object, "A"(optional))

The function writes an object to a new text file. If the file already exists, it is truncated. If the folder does not exist, it will create the folder.

If Object is a string or a number, it is written to the new text file. If Object is an array, each cell is written as a separate line. If Object is a graph, first line is the graph name followed by one number per line.

If the optional "A" character is used, the file is written in ANSI format. Otherwise, default Unicode format is used. For example,

```
arr = {   "[Diode]",
          "Forward Voltage = 1.2V",
          "Forward Current = 30mA",
          "",
          "[Mosfet]",
          "Forward Voltage = 0.7V",
          ""}
FileWrite("C:\powersim\data\config.ini", arr);
```

The resulting file is:

```
[Diode]
Forward Voltage = 1.2V
Forward Current = 30mA
[Mosfet]
Forward Voltage = 0.7V
```

GetLocal(Name)

The function returns a value that is specific to the local computer.

The parameter *Name* can have the following options:

Name	Return value
PSIMPATH	Full PSIM folder path, for example, "C:\Powersim\PSIM11.0.2\"
PARAMPATH	Full folder path of the parameter file, for example, "C:\Data\"
SCHPATH	Full folder path of the schematic file, for example, "C:\Schematic files\". If this value is used in a Parameter tool window, it returns an empty string.
SCHNAME	File name of the schematic file including extension, for example, "file1.psimsch". If this value is used in a Parameter tool window, it returns an empty string.

Graph Functions

The following curve data type and file functions are provided:

Curve data type

Curve data is a set of numbers that represents the behavior of one variable within a specific boundary. Curve data consists of:

- Curve name
- Array of real numbers.
- Array of validation flags, one flag for each number in the above array.

BodePlot(PlotName, X-Axis, Curve1, Curve2, ArrayOfCurves, …)

The function plots two graph windows. The first window displays 20 * log10 of magnitude of complex number arrays, and the second window displays the angle of complex number arrays.

Function parameters are:

PlotName: Name of the plot. It is a string.

X-Axis, Curve1, Curve2: They can be a Curve or Array of numbers. Unlimited numbers of curves can be shown together. If any curves including X are complex

numbers, real and imaginary parts are shown as separate curves. Arrays containing multiple curves or array of numbers can be placed anywhere in the list of parameters.

GetCurve(ArrayOfCurves, CurveName)

The function searches for CurveName in ArrayOfCurves and returns the curve. For example,

```
str = GetCurve(graph, "V1");
```

GetCurveName(Curve1)

The function returns the name of a curve. For example,

```
str = GetCurveName(Curve1);
```

GraphMerge(OutputFileName, InputFile1, InputFile2, InputFile3, …)

This function merges two or more SIMVIEW graph files and saves the result in File OutputFileName. Its functionality is identical to SIMVIEW's Merge function.

All the function parameters must be fully qualified file path. Use the function GetLocal() to build a fully qualified file path that starts from the Parameter file path 'GetLocal(PARAMPATH)' or from the schematic file path 'GetLocal(SCHPATH)'. For example,

```
OutputFile = GetLocal(PARAMPATH) + "mergedGraph.smv";
File1 = GetLocal(PARAMPATH) + "graph1.smv";
File2 = GetLocal(PARAMPATH) + "graph2.smv";
GraphMerge(OutputFile, File1, File2);
```

All input files must be valid SIMVIEW .smv or .txt files.

GraphRead(FilePath)
GraphRead(FilePath, "CurveName1", "CurveName2", …)

The function GraphRead(FilePath) reads the entire SIMVIEW file with .smv or .txt extension and places the content in an array of curves. The return value is an array of curves. The first cell is the X-Axis and subsequent cells are the rest of curves in file. For example, if MyGraph.smv file contains the following:

```
Time  V1  V2
0.001  1.1   2.2
0.002  3.3   4.4
… …
```

With the function call below:

```
arr = GraphRead("C:\Powersim\MyGraph.smv");
```

arr[0] will contain the first column (arr[0] = {0.001 0.002 …}), and arr[1] will contain the second column (arr[1] = {1.1 3.3 …}).

In the case of a large data file, GraphRead(FilePath) will return oversampled data, not the full data. A large data file is defined such that when it is read, it will result in less than 1GB of physical RAM memory left in the computer. To load the full data in this case, use the function GraphRead(FilePath, "CurveName1", "CurveName2", …) to load only the selected curves.

GraphWrite(FilePath, X-Axis, Curve1, Curve2, ArrayOfCurves, …)

The function writes a SIMVIEW graph file. If FilePath's extension is .txt, it will be write a text file. Otherwise binary .smv format file is written.

Function parameters are:

PlotName:	Name of the plot. It is a string.
X-Axis, Curve1, Curve2:	They can be a Curve or Array of numbers. Unlimited numbers of curves can be written together. If any curves including X are complex numbers, real and imaginary parts are written as separate columns. Arrays containing multiple curves or array of numbers can be placed anywhere in the list of parameters.

MakeCurve(CurveName, NumberOfRows)
MakeCurve(CurveName, ArrayOfValues)
MakeCurve(CurveName, NumberOfRows, Formula, VarName1, ArrayOfValues, VarName2, StartValue, Increment …)

The function creates a new curve data.

Function parameters are:

CurveName:	Name of the curve
NumberOfRows:	It is not possible to resize a Curve therefore it is important that correct number is used here.
ArrayOfValues:	Size of array is used as number of rows and graph is initialized to the array values.
Formula (optional).	This formula is used to initialize the curve. Variables in the formula are defined either as array of values or a start value and increment value. To set a variable to a constant, one must use the constant as the start value and zero as increment. Any variable in the formula that is not defined as parameter, is taken from the main script.

Plot(PlotName, X-Axis, Curve1, Curve2, ArrayOfCurves, …)

This function plots a graph.

Function parameters are:

PlotName:	Name of the plot. It is a string.
X-Axis, Curve1, Curve2:	They can be a Curve or Array of numbers. Unlimited numbers of curves can be shown together. If any curves including X are complex numbers, real and imaginary parts are shown as separate curves.

Arrays containing multiple curves or array of numbers can be placed anywhere in the list of parameters.

The example “plot sawtooth and triangular.script”, available in the “examples\Script” folder, plots the sawtooth and triangular waveforms.

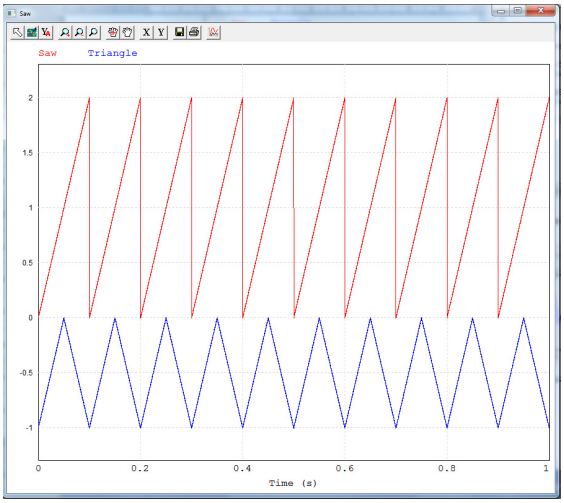
```
// Plotting sawtooth and triangular waveforms
ScriptOption("NoLog*");

t=MakeCurve("Time", Array(100u, 1, 100u)); // t = {1e-4, 2e-4, 3e-4, ..., 1}
Saw1 = ((t / 50m) % 2);
SetCurveName(Saw1, "Saw");

Saw2 = -abs(Saw1-1);
SetCurveName(Saw2, "Triangle");

Plot("Saw", t, Saw1, Saw2);
```

Running the script will generate the waveforms below:



SetCurveName(Curve1, NewName)

The function sets a new name for the curve. For example,

```
SetCurveName(Curve1, "XAmplitude");
```

Simulation Functions

The following simulation functions are provided:

ASimulate(SchematicFilePath, SimviewFilePath, SimulationOptions)

This function is the same as 'Simulate' except that it returns results immediately while simulation is running in parallel in a separate thread. If the graph produced by this simulation is needed for calculations later, then this function should not be used.

Simulate(SchematicFilePath, SimviewFilePath, SimulationOptions, ReturnGraph)
SimulateDSIM(SchematicFilePath, SimviewFilePath, SimulationOptions, ReturnGraph)
SimulateLT(SchematicFilePath, SimviewFilePath, SimulationOptions, ReturnGraph)

The function Simulate opens the schematic file and runs the simulation using the PSIM engine. The function SimulateDSIM runs the simulation using the DSIM engine. The function SimulateLT runs the simulation using LTspice.

The function does not return until simulation is over. Only the parameter "SchematicFilePath" is mandatory. All other parameters are optional.

The parameter "SimviewFilePath" defines the simulation output file. For Simulate, SimulateDSIM, and SimulateSpice, If the file has the .txt extension, the output file will be in text format. If the file has the .smv extension, the output file will be in binary format. For SimulateLT, the output file will be in binary format with the .raw extension regardless.

The parameter "ReturnGraph" must be a variable that is set to an array of curves. In case of ac sweep simulation, 'ReturnGraph' will be the ac sweep graph.

SimulationOptions is a string, and it must be enclosed in "". Valid options are:

TotalTime
TimeStep
PrintTime
PrintStep
SaveFlag
LoadFlag

In addition, normal variables can be added here too. For example,

```
Simulate ("C:\Powersim\files\active filter.psimsch", "C:\Powersim\Graphs\active filter.txt",
"TotalTime=100m; TimeStep=10u; PrintTime=10m; PrintStep=2; SaveFlag=1; LoadFlag=0; R1=11K;",
graph1);
```

In addition to the variable list, any variables that come before the Simulate function are passed to the schematic.

Simview(FileName, Curve1(optional), Curve2(optional), Curve3(optional), …)

This function runs SIMVIEW (if it's not already running) and opens the file "FileName". The file name must be a fully qualified file path of a valid SIMVIEW file. If optional Curve names are included in the parameter list, these curves will be displayed. For example:

```
File1 = GetLocal(PARAMPATH) + "graph1.smv";
Simview(File1, "vload");
```

The example "simulation control example.script", available in the "examples\script" folder, shows how one can define the time step and total time, run the simulation, perform calculation, and save the result back to a file, and plot waveforms.

```
// This example does the following:
//   - Define time step and total time
//   - Use the defined time step and total time to simulate the circuit sch1.psimsch
//   - Create a sine wave V_sine
//   - Plot V3 from the simulation result plus V_sine
//   - Write the original simulation result plus V_sine to a new file
Step_t  =  1E-5;     // define simulation time step
Total_t = 0.1;        // define simulation total time

//t = Array(1E-5, .1, 1E-5);
t = Array(Step_t, Total_t, Step_t);

File1 = GetLocal("PARAMPATH") + "Files\sch1.psimsch";
smvFile = GetLocal("PARAMPATH") + "Files\graph1.txt";

// Simulate the circuit
Simulate(File1, smvFile, TimeStep = Step_t, TotalTime=Total_t, graph1);
Count = sizeof(graph1);   // obtain simulation result
V3 = GetCurve(graph1, "v3");   // obtain v3 curve
V4 = GetCurve(graph1, "v4");   // obtain v4 curve

// Create a sine wave
V_sine = 110 * sin( (t*60*2*pi) + (60/180*pi)); //  Sin wave: 60Hz, 60 degrees shift

// Plot V3 and V_sine vs. t
Plot("DD", t, V3, V_sine);
GraphWrite(GetLocal("PARAMPATH") + "Files\Files\graph2.smv", graph1, V_sine);
```

Please refer to many other examples in the "examples\script" folder that illustrate various tasks that script can perform.

Other

The following functions are provided for the convenience of using the Script Tool:

Print(item1, item2, ...)

Prints out multiple items in the log area. All in one line. One can use this function to display the values of an array, for example, “Print(array_a)”.

Clear()

Clears the log area. This function has no parameters.

LoadScript(FilePath, (optional)inputDictionary, (optional)var1=value1, ...)

Loads an external script and returns a Dictionary object containing all variables in the script.

If script 'param1.script' contains:

```
a = 2;
b = 5;
```

then:

```
dict1 = LoadScript("param1.script");
```

returns a dictionary object 'dict1' containing the values of a and b

```
s1 = dict1["a"];
s2 = dict1["b"];
```

If script requires input values, we can provide it either by pairs of variable name/value or by an input dictionary object.

MessageBox(text1, text2, ..., TITLE="", BUTTONS="", ICON="")

Show a dialog box with a text message. User must click a button to continue. Return value is a text, indicating the button that was pressed.

Return value is a text describing the button that was pressed. Possible return values:

"OK"
"CANCEL"
"YES"
"NO"
"CONTINUE"
"RETRY"
"TRYAGAIN"

Following line will show a MessageBox with the text "This dialog was shown 15 time(s).":

```
num = 15;
```

```
    result = MessageBox("This dialog was shown ", num, " time(s).", TITLE="Testing buttons",
BUTTONS=CANCELTRYAGAINCONTINUE, ICON=STOP);
```

Texts and variables will be concatenated to form a single text and shown in the messagebox. There are three special optional parameters: TITLE, BUTTONS and ICON. TITLE is the title of the dialog box. Acceptable values for BUTTONS and ICON are as follows:

Accepted values for BUTTONS:

OK	OK button. (Default)
OKCANCEL	OK and Cancel buttons
YESNO	Yes and No buttons
YESNOCANCEL	Yes, No and Cancel buttons
CANCELTRYAGAINCONTINUE	Cancel, Try again and Continue buttons
RETRYCANCEL	Retry and Cancel buttons

Accepted values for ICON:

INFORMATION
QUESTION
STOP
WARNING

ScriptOption("Nolog") This will disable the variable value display.
ScriptOption("log") This will enable the variable value display.

The example below shows how the function is used:

```
a = 1;
ScriptOption("Nolog");  // disable display
b = 2;
ScriptOption("log");              // enable display
c = a + b;
```

The script display window will show:

a = 1
c = 3

without displaying the value of b.

Error{"text %f, %f", var1, var2}

Error statement

Warning{"text %f, %f", var1, var2}

Warning statement

An example of the error and warning statement is shown below.

```
a = 1;
if (a > 2)
```

```
{
  Error("Error: Variable a is greater than 2. a = %f", a);
}
else if (a < 0)
{
  Warning("Warning: Variable a is less than 0. a = %f", a);
}
```

IsNull
IsEmpty

Checks if the parameter is Null or Empty

```
IsNull(data);
IsEmpty(data);
```

TypeName(var)

Returns the name of the type of data stored in a variable.

```
TypeName(var);
```

Return value is text. Possible return values:

""	Null or Empty
"REAL"	Real number
"INT"	Integer
"STRING"	Text
"COMPLEX"	Complex number
"ARRAY"	Array
"ARRAY_R"	Array of real numbers
"DICTIONARY"	Dictionary object
"FORMULA"	Formula
"FUNCTION"	Pointer to a function
"CURVE"	A single graph curve
"XML"	XML object
"POINTER"	Pointer to a generic object
"DOCUMENT"	Pointer to a schematic file
"PSIMPART"	A single schematic element

Examples:

```
Var1 = "AppleOrange";
Var2 = 15.2;
Var3 = 88;
s1 = TypeName(Var1); // returns "STRING"
s2 = TypeName(Var2); // returns "REAL"
s3 = TypeName(Var3); // returns "REAL"
```