Section 10
System tasks and functions

Proposal for additional of Interpolation Function System Task

--Martin O'Leary, Cadence Design Systems, 5/21/2004

NOTE: changes are all in section 10.9 which is a totally new section

Rational: The interpolation capability is useful for quickly making models from measured data and also for creating more efficient models of complex transistor level circuits.

This section describes system tasks and functions available in Verilog-AMS HDL.

10.1 Environment parameter functions

The syntax for these functions are shown in Syntax 10-1.

```
environment_parameter_functions ::=
        $temperature
      | $abstime
      | $realtime [ ( real_number ) ]
      | $vt [ ( temperature_expression ) ]
```

Syntax 10-1—Syntax for the environment parameter functions

These functions return information about the current environment parameters as a real value.

$temperature does not take any input arguments and returns the circuit's ambient temperature in Kelvin units.

$abstime returns the absolute time, that is a real value number representing time in seconds.

$realtime can have an optional argument which scales the time. If no argument is given, **$realtime**'s return value is scaled to the **'time_unit** of the module which invoked it. If an argument is given, **$realtime** shall divide the absolute time by the value of the argument (i.e., scale to the value specified in the argument). The argument for **$realtime** follows the symantics of the **'time_unit**, that is it shall consist of an integer followed by a scale

factor. Valid integers are: `1`, `10`, and `100`; valid scale factors are: `s` (seconds), `ms` (milliseconds), `us` (microseconds), `ns` (nanoseconds), `ps` (picoseconds), and `fs` (femtoseconds).

$vt can optionally have temperature (in Kelvin units) as an input argument and returns the thermal voltage (*kT/q*) at the given temperature. **$vt** without the optional input temperature argument returns the thermal voltage using **$temperature**.

Note: Previous Verilog-A modules using **$realtime** may not produce the same results as in the past as `$realtime` used to return time scaled to one (`1`) second. If the `time_unit` of the **'timescale** directive is set to `1s`, the behavior shall be the same. For analog blocks, the **$abstime** function should typically be used, as it returns time in seconds.

10.2 $random function

The syntax for this function is shown in Syntax 10-2.

```
random_function ::=
    $random [ ( seed_expression ) ] ;
```

Syntax 10-2—Syntax for the random_function

$random provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative.

seed_expression controls the numbers **$random** returns. *seed* shall be a **reg**, **integer**, or **time** variable. The *seed* value shall be assigned to this variable prior to calling **$random**.

Examples:

Where `b > 0`, the expression `($random % b)` gives a number in the following range: [(`-b+1`) : (`b-1`)].

The following code fragment shows an example of random number generation between `-59` and `59`:

```
integer rand;
rand = $random % 60;
```

10.3 $dist_ functions

The syntax for these functions are shown in Syntax 10-3.

```
distribution_functions ::=
     $digital_dist_functions ( args ) ;
    | $rdist_uniform ( seed, start_expression, end_expression ) ;
    | $rdist_normal ( seed, mean_expression, standard_deviation_expression ) ;
    | $rdist_exponential ( seed, mean_expression ) ;
    | $rdist_poisson ( seed, mean_expression ) ;
    | $rdist_chi_square ( seed, degree_of_freedom_expression ) ;
    | $rdist_t ( seed, degree_of_freedom_expression ) ;
    | $rdist_erlang ( seed, k_stage_expression, mean_expression ) ;

seed ::=
    integer_variable_identifier
```

Syntax 10-3—Syntax for the probabilistic distribution functions

The following rules apply to these functions.

- All parameters to the system functions are real values, except for *seed* (which is defined by **$random**()). For the **$rdist_exponential**, **$rdist_poisson**, **$rdist_chi_square**, **$rdist_t**, and **$rdist_erlang** functions, the parameters *mean*, *degree_of_freedom*, and *k_stage* shall be greater than zero (`0`).
- Each of these functions returns a pseudo-random number whose characteristics are described by the function name, e.g., **$rdist_uniform** returns random numbers uniformly distributed in the interval specified by its parameters.
- For each system function, the *seed* parameter is an `inout` parameter; that is, a value is passed to the function and a different value is returned. The system functions shall always return the same value given the same *seed*. This facilitates debugging by making the operation of the system repeatable. The argument for *seed* shall be an integer variable, which is initialized by the user and only updated by the system function. This ensures the desired distribution is achieved.
- All functions return a real value.
- In **$rdist_uniform**, the *start* and *end* parameters are real inputs which bound the values returned. The *start* value shall be smaller than the *end* value.
- The *mean* parameter used by **$rdist_normal**, **$rdist_exponential**, **$rdist_poisson**, and **$rdist_erlang** is an real input which causes the average value returned by the function to approach the value specified.
- The *standard_deviation* parameter used by **$rdist_normal** is a real input, which helps determine the shape of the density function. Using larger numbers for *standard_deviation* spreads the returned values over a wider range. Using a *mean* of zero (`0`) and a *standard_deviation* of one (`1`), **$rdist_normal** generates Gaussian distribution.

- The *degree_of_freedom* parameter used by **$rdist_chi_square** and **$rdist_t** is a real input, which helps determine the shape of the density function. Using larger numbers for *degree_of_freedom* spreads the returned values over a wider range.

10.4 Simulation control system tasks

There are two simulation control system tasks, **$finish** and **$stop**.

10.4.1 $finish

The syntax for this task is shown in Syntax 10-4.

```
finish_task ::=
    $finish [ ( n ) ] ;
```

Syntax 10-4—Syntax for the finish_task

$finish simply makes the simulator exit. If an expression is supplied to this task, its value determines which diagnostic messages are printed before the prompt is issued, as shown in Table 10-1. One (`1`) is the default if no argument is supplied.

Table 10-1—Diagnostic messages

Parameter	Message
0	Prints nothing
1	Prints simulation time and location
2	Prints simulation time, location, and statistics about the memory and CPU time used in simulation

10.4.2 $stop

The syntax for this task is shown in Syntax 10-5.

```
stop_task ::=
    $stop [ ( n ) ] ;
```

Syntax 10-5—Syntax for the stop_task

$stop causes simulation to be suspended at a converged timepoint. This task takes an optional expression argument (`0`, `1`, or `2`), which determines what type of diagnostic message is printed. The amount of diagnostic messages output increases with the value of *n*, as shown in Table 10-1.

10.5 File operation tasks

This section details the file operation tasks.

10.5.1 $fopen

The syntax for this task is shown in Syntax 10-6.

```
file_open_task ::=
    integer multi_channel_descriptor = $fopen ( " file_name " ) ;
```

Syntax 10-6—Syntax for the file_open_task

$fopen opens the file specified as an argument and returns a 32-bit multichannel descriptor which is uniquely associated with the file. It returns `0` if the file could not be opened for writing.

The multichannel descriptor can be thought of as a set of 32 flags, where each flag represents a single output channel. The least significant bit (`bit 0`) of a multichannel descriptor always refers to the standard output. The standard output is also called `channel 0`. The other bits refer to channels which have been opened by **$fopen**.

The first call to **$fopen** opens `channel 1` and returns a multichannel descriptor value of `2`—that is, `bit 1` of the descriptor is set. A second call to **$fopen** opens `channel 2` and returns a value of `4`—that is, only `bit 2` of the descriptor is set. Subsequent calls to **$fopen** open `channels 3`, `4`, `5`, and so on and return values of `8`, `16`, `32`, and so on, up to a maximum of 32 open channels. Thus, a channel number corresponds to an individual bit in a multichannel descriptor.

10.5.2 $fclose

The syntax for this task is shown in Syntax 10-7.

```
file_close_task ::=
    $fclose ( multi_channel_descriptor_identifier ) ;
```

Syntax 10-7—Syntax for the file_close_task

$fclose closes the channels specified in the multichannel descriptor and does not allow any further output to the closed channels. **$fopen** reuses channels which have been closed.

10.6 Display tasks

The syntax for these functions are shown in Syntax 10-8.

```
display_tasks ::=
        $strobe ( list_of_arguments ) ;
      | $display ( list_of_arguments ) ;
      | $monitor ( list_of_arguments ) ;
      | $write ( list_of_arguments ) ;
```

Syntax 10-8—Syntax for the display_tasks

The following rules apply to these functions.

— **$strobe** provides the ability to display simulation data when the simulator has converged on a solution for all nodes.
— **$strobe** displays its arguments in the same order they appear in the argument list. Each argument can be a quoted string, an expression which returns a value, or a null argument.
— The contents of string arguments are output literally, except when certain escape sequences are inserted to display special characters or specify the display format for a subsequent expression.
— Escape sequences are inserted into a string in three ways:

- The special character \ indicates the character to follow is a literal or non-printable character (see Table 10-2).
- The special character `%` indicates the next character shall be interpreted as a format specification which establishes the display format for a subsequent expression argument (see Table 10-3). For each `%` character which appears in a string, a corresponding expression argument shall be supplied after the string.
- The special character string `%%` indicates the display of the percent sign character (`%`) (see Table 10-2).

— Any null argument produces a single space character in the display. (A *null argument* is characterized by two adjacent commas (`,,`) in the argument list.)
— When **$strobe** is invoked without arguments, it simply prints a newline character.

The **$display** task provides the same capabilities as **$strobe**. The **$write** task provides the same capabilities as **$strobe**, but with no newline. The **$monitor** task provides the same capabilities as **$strobe**, but outputs only when a parameter changes.

10.6.1 Escape sequences for special characters

The escape sequences shown in Table 10-2, when included in a string argument, print special characters.

Table 10-2— Escape sequences for printing special characters

`\n`	The newline character
`\t`	The tab character
`\\`	The \ character
`\"`	The `"` character
`\ddd`	A character specified by 1 to 3 octal digits
`%%`	The `%` character

10.6.2 Format specifications

Table 10-3 shows the escape sequences used for format specifications. Each escape sequence, when included in a string argument, specifies the display format for a subsequent expression. For each `%` character (except `%m` and `%%`) which appears in a string, a corresponding expression shall follow the string in the argument list, except a null argument. The value of the expression replaces the format specification when the string is displayed.

Table 10-3— Escape sequences for format specifications

`%h` or `%H`	Display in hexadecimal format
`%d` or `%D`	Display in decimal format
`%o` or `%O`	Display in octal format
`%b` or `%B`	Display in binary format
`%c` or `%C`	Display in ASCII character format
`%m` or `%M`	Display hierarchical name
`%s` or `%S`	Display as a string

Any expression argument which has no corresponding format specification is displayed using the default decimal format in **$strobe**.

The format specifications in Table 10-4 are used for real numbers and have the full formatting capabilities available in the C language. For example, the format specification `%10.3g` sets a minimum field width of 10 with three (3) fractional digits.

Table 10-4— Format specifications for real numbers

`%e` or `%E`	Display 'real' in an exponential format
`%f` or `%F`	Display 'real' in a decimal format
`%g` or `%G`	Display 'real' in exponential or decimal format, whichever format results in the shorter printed output

10.6.3 Hierarchical name format

The `%m` format specifier does not accept an argument. Instead, it causes the display task to print the hierarchical name of the module, task, function, or named block which invokes the system task containing the format specifier. This is useful when there are many instances of the module which call the system task. One obvious application is timing check messages in a flip-flop or latch module; the `%m` format specifier pinpoints the module instance responsible for generating the timing check message.

10.6.4 String format

The `%s` format specifier is used to print ASCII codes as characters. For each `%s` specification which appears in a string, a corresponding parameter shall follow the string in the argument list. The associated argument is interpreted as a sequence of 8-bit hexadecimal ASCII codes, with each 8 bits representing a single character. If the argument is a variable, its value shall be right-justified so the right-most bit of the value is the least-significant bit of the last character in the string. No termination character or value is required at the end of a string and leading zeros (`0`) are never printed.

10.7 Announcing discontinuity

The **$discontinuity** function is used to give hints to the simulator about the behavior of the module so the simulator can control its simulation algorithms to get accurate results in exceptional situations. This function does not directly specify the behavior of the module. **$discontinuity** shall be executed whenever the analog behavior changes discontinuously.

The general form is

$discontinuity`[ ( constant_expression ) ];`

where *constant_expression* indicates the degree of the discontinuity. `$discontinuity(i)` implies a discontinuity in the *i*'th derivative of the constitutive equation with respect to either a signal value or time where *i* must be non-negative. Hence, `$discontinuity(0)` indicates a discontinuity in the equation, `$discontinuity(1)` indicates a discontinuity in its slope, etc.

Note: Because discontinuous behavior can cause convergence problems, discontinuity shall be avoided whenever possible.

The filter functions (**transition()**, **slew()**, **laplace()**, etc.) can be used to smooth discontinuous behavior. However, in some cases it is not possible to implement the desired functionality using these filters. In those cases, the **$discontinuity** function shall be executed when the signal behavior changes abruptly.

Note: Discontinuity created by switch branches and built-in system functions, such as **transition()** and **slew()**, does not need to be announced.

Examples:

Example 1

The following example uses the discontinuity function to model a relay.

```
module relay (c1, c2, pin, nin) ;
    inout c1, c2 ;
    input pin, nin ;
    electrical c1, c2, pin, nin ;
    parameter real r=1 ;

    analog begin
        @(cross(V(pin,nin))) $discontinuity ;
        if (V(pin,nin) >= 0)
            I(c1,c2) <+ V(c1,c2)/r;
        else
            I(c1,c2) <+ 0 ;
    end
endmodule
```

In this example, **cross()** controls the time step so the time when the relay changes position is accurately resolved. It also triggers the **$discontinuity** function, which causes the simulator to react properly to the discontinuity. This would have been handled automatically if the type of the branch `(c1,c2)` had been switched between voltage and current.

Example 2

Another example is a source which generates a triangular wave. In this case, neither the model nor the waveforms generated by the model are discontinuous. Rather, the waveform generated is piecewise linear with discontinuous slope. If the simulator is aware of the abrupt change in slope, it can adapt to eliminate problems resulting from the discontinuous slope (typically changing to a first order integration method).

```
module triangle(out);
    output out;
    voltage out;
    parameter real period = 10.0, amplitude = 1.0;
    integer slope;
    real offset;

    analog begin
        @(timer(0, period)) begin
            slope = +1;
            offset = $abstime ;
            $discontinuity;
        end

        @(timer(period/2, period)) begin
            slope = -1 ;
            offset = $abstime;
            $discontinuity ;
        end
```

```
        V(out) <+ amplitude*slope*
            (4*($abstime - offset)/period - 1);
    end
endmodule
```

10.8 Time related functions

The **$bound_step()** function puts a bound on the next time step. It does not specify exactly what the next time step is, but it bounds how far the next time point can be from the present time point. The function takes the maximum time step as an argument. It does not return a value.

The general form is

```
$bound_step ( expression );
```

where *expression* is a required argument and represents the maximum timestep the simulator can advance.

Examples:

The example below implements a sinusoidal voltage source and uses the **$bound_step()** function to assure the simulator faithfully follows the output signal (it is forcing 20 points per cycle).

```
module vsine(out);
    output out;
    voltage out;
    parameter real freq=1.0, ampl=1.0, offset=0.0;

    analog begin
        V(out) <+ ampl*sin(2.0*`M_PI*freq*$abstime) + offset;
        $bound_step(0.05/freq);
    end
endmodule
```

For details on the **last_crossing()** function, see 4.4.10.

10.9 Interpolation Function

The **$table_model** function models the behavior of a system by interpolating between data points that are samples of that system's behavior. To build such a model, the user needs to provide a set of sample behavior points $(x_{i1}, x_{i2}, .., x_{iN}, y_i)$ so that $f(x_{i1}, x_{i2}, .., x_{iN}) = y_i$, where f is the model function and N is the number of independent variables of the model. Using interpolation techniques, one can get the model value at any point in the domain of the sample points approximately. In the case when the evaluated point is outside the domain of the sample points, extrapolation techniques can be used the

calculate the model value at that point. However, extrapolation is usually unreliable when the evaluated point is far from the domain of the sample points.

The syntax for the table_model function is shown in Syntax 10-9.

```
table_model_function ::=
    $table_model ( table_inputs, table_source, table_control_string )

table_inputs::=
    expr{, expr}

table_source ::=
    filename_string | table_model_array

table_model_array::=
    1st_dim_array_identifier [, 2nd_dim_array_identifier [, 3rd_dim_array_identifier]]],
output_array_identifier | multi_dimensional_array_identifier

table_control_string :: = "[1st_dim_table_sub_ctrl_string] [, 2nd_dim_table_sub_ctrl_string [,
3rd_dim_table_sub_ctrl_string]]"

table_sub_ctrl_string ::=
    [table_degree_char] [table_extrap_char]

table_degree_char ::=
    1 | 2 | 3

table_extrap_char ::=
    C | L | S | E
```

Syntax 10-9—Syntax for table model function

10.9.1 Table Model Inputs

`table_inputs` are numerical expressions that are used as the independent model variables. They can be any legal expressions that can be assigned to an analog signal.

10.9.2 Data Source

`data_source` specifies the source of sample points for the table model. There are two kinds of data sources: files and arrays. The file source requires that the sample points be stored in a file, while the array source requires that the data points be stored in a set of array variables. The user can select the data source by either providing the file name of a file source or a set of array variables.

For a file source, conceptually the sample points are stored in the following format:

$P_1 P_2 P_3 ... P_M$

where P_i (i =$1...M$) are the sample points. Each sample point P_i is represented as a sequence of numbers in the order of X_{i1} X_{i2} .. X_{iN} Y_i, where X_{ik} is the coordinate of the sample point in k'th dimension and Y_i is the model value at this sample point. Each

sample point must be separated by a newline. To increase readability of the data file, comments can occur in any place of the file. Comments begin with '#' and end with a newline.

The sample points can be stored in the file in any order. However, the lexical ascending order is recommended for performance reason. The lexical order $P_i < P_j$ holds if and only if there exists k, so that $X_{il} = X_{jl}$ $(l = 1...k-1)$ and $X_{ik} < X_{jk}$. If the sample points are not required to be any particular order. The following shows an sample data file:

```
# example.tbl
# 2-D table model sample example
#
#       x       y       f(x,y)
        -10     -10     0
        -10     -8      -0.4
        -10     -6      -0.8
        -9      -10     0.2
        -9      -8      -0.2
        -9      -6      -0.6
        -9      -4      -1
        -8      -10     0.4
        -8      -9      0.2
        -8      -7      -0.2
        -8      -5      -0.6
        -8      -3      -1
        -7      -10     0.6
        -7      -9      0.4
        -7      -8      0.2
        -7      -7      0
        -7      -6      -0.2
        -7      -5      -0.4
```

If the user choose the array source, a set of one-dimensional arrays or a single multi-dimensiona array that contains the data points should be passed to the table_model function. The size of these arrays is the number of sample points in the table, M. The data will be stored in the arrays such that for the kth dimension of the ith sample point, `kth_dim_array_identifier`$[i] = X_{ik}$ and such that for the ith sample point `output_array_identifier`$[i] = Y_i$.

If a multi-dimensional array, that array must be two dimensional. The first dimension is designated for points and the second dimension is is designated for coordinates of the point and its values. Suppose p is such an array, $p[i][j]$ contains the value of the j'th coordinate of i'th point when $0 \le j < N$ and $p[i][N]$ contains the value of the sample point, where N is the dimension of the table_model.

10.9.3 Control String

The control string is used to control the numerical aspects of the interpolation process. It consists of subcontrol strings for each dimension of the model. The subcontrol string

may contain one degree character and one or two extrapolation method characters. The degree is a digit representing the degrees of the splines used for the interpolation. The degree should not exceed 3. When the degree character does not appear in the subcontrol string, degree 1 (i.e. linear interpolation) is assumed.

The extrapolation method controls how the point is evaluated when the point is beyond the region of the user provided sample points. Three extrapolation methods are supported (see Figure 10.1). The Clamp extrapolation method uses a horizontal line that passes through the nearest sample point, also called the end point, to extend the model evaluation. The Linear extrapolation method models the extrapolation through a tangent line at the end point. The Spline extrapolation method uses the polynomial for the nearest segment (the segment at the end) to evaluate a point beyond the interpolation area. The user can also disable extrapolation by choosing the Error extrapolation method. In this case, when the system tries to evaluate a point beyond the interpolation region, an interpolation error will be reported. The extrapolation method characters used to specify the extrapolation methods are shown in Table 10-5.

Table 10-5—Meaning of $table_model extrapolation character

extrapolation character	meaning of the character
C	Clamp extrapolation
L	Linear extrapolation (default)
S	Spline extrapolation
E	Error extraplation

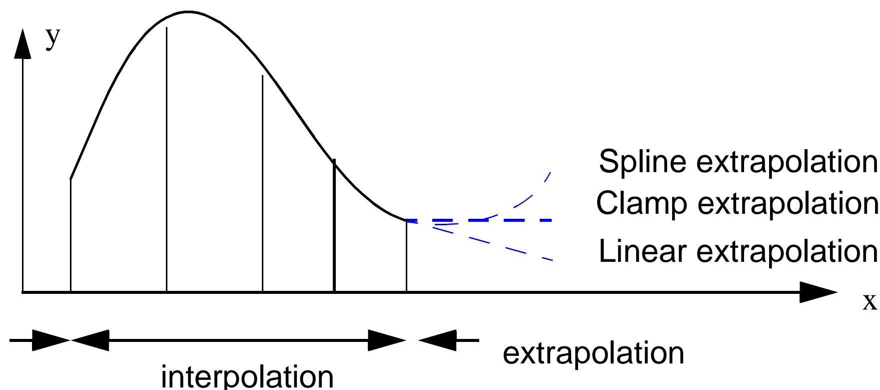

Figure 10 Using the CLAMP and LINEAR end conditions

Users can use up to 2 extrapolation method characters to specify the extrapolation method used for each end. When no extrapolation method character is given, the Linear extrapolation method will be used for both ends as default. When one extrapolation

method character is given, the specified extrapolation method will be used for both ends. When two extrapolation method characters are given, the first character specifies the extrapolation method used for the end with smaller coordinate value, and the second character is used for the end with larger coordinate value.

Table 10-6 contains some examples of the control strings and interpretation.

Table 10-6—Example control strings and their interpretation

control string	1st dimension interpolation method	1st dimen. left extrap. method	1st dimen. right extrap. method	2nd dimen. interp. method	2nd dimen. left extrap. method	2nd dimen. right extrap. method
“1CL,3SL”	1	Clamp	Linear	3	Spline	Linear
“1CL,3SL”	1	Clamp	Linear	3	Spline	Linear
“1L,3S”	1	Linear	Linear	3	Spline	Spline
“1,3CL”	1	Linear	Linear	3	Clamp	Linear
“1”	1	Linear	Linear	1	Linear	Linear
“,3”	1	Linear	Linear	3	Linear	Linear
“”	1	Linear	Linear	1	Linear	Linear

10.9.4 Examples

The following examples are for the same 2-D table model but using the different types of data source.

Example 1: $table_model with file data source

Note the contents of example.tbl are those given earlier in this section.

```
module measured_resistance (a, b);
electrical  a, b;
inout       a, b;

analog  begin
    I(a, b) <+ $table_model (V(a), V(b),
“example.tbl", "3S,1S");
end
endmodule
```

Example 2: *$table_model with data source in one-dimensional arrays*

```
module measured_resistance (a, b);
electrical  a, b;
inout       a, b;
real x[0:17], y[0:17], f_xy[0:17];

analog begin
    @(initial_step) begin
        x[0]= -10; y[0]=-10;  f_xy[0]=0;    // 0th sample point
        x[1]= -10; y[1]=-8;   f_xy[1]=-0.4; // 1st sample point
        x[2]= -10; y[2]=-6;   f_xy[2]=-0.8; // 2nd sample point
        x[3]= -9;  y[3]=-10;  f_xy[3]=0.2;
        x[4]= -9;  y[4]=-8;   f_xy[4]=-0.2;
        x[5]= -9;  y[5]=-6;   f_xy[5]=-0.6;
        x[6]= -9;  y[6]=-4;   f_xy[6]=-1;
        x[7]= -8;  y[7]=-10;  f_xy[7]=0.4;
        x[8]= -8;  y[8]=-9;   f_xy[8]=0.2;
        x[9]= -8;  y[9]=-7;   f_xy[9]=-0.2;
        x[10]= -8; y[10]=-5;  f_xy[10]=-0.6;
        x[11]= -8; y[11]=-3;  f_xy[11]=-1;
        x[12]= -7; y[12]=-10; f_xy[12]=0.6;
        x[13]= -7; y[13]=-9;  f_xy[13]=0.4;
        x[14]= -7; y[14]=-8;  f_xy[14]=0.2;
        x[15]= -7; y[15]=-7;  f_xy[15]=0;
        x[16]= -7; y[16]=-6;  f_xy[16]=-0.2;
        x[17]= -7; y[17]=-5;  f_xy[17]=-0.4;
    end
    I(a, b) <+ $table_model (V(a), V(b), x, y, f_xy, "3S,1S");
end
endmodule
```

Example 3: *$table_model with data source in a multi-dimensional array*

```
module measured_resistance (a, b);
electrical  a, b;
inout       a, b;
real d[0:17][0:2]; // data source

analog begin
    @(initial_step) begin
        d[0][0]= -10; d[0][1]=-10;  d[0][2]=0;    // 0th sample point
        d[1][0]= -10; d[1][1]=-8;   d[1][2]=-0.4; // 1st sample point
        d[2][0]= -10; d[2][1]=-6;   d[2][2]=-0.8; // 2nd sample point
        d[3][0]= -9;  d[3][1]=-10;  d[3][2]=0.2;
        d[4][0]= -9;  d[4][1]=-8;   d[4][2]=-0.2;
        d[5][0]= -9;  d[5][1]=-6;   d[5][2]=-0.6;
        d[6][0]= -9;  d[6][1]=-4;   d[6][2]=-1;
        d[7][0]= -8;  d[7][1]=-10;  d[7][2]=0.4;
        d[8][0]= -8;  d[8][1]=-9;   d[8][2]=0.2;
        d[9][0]= -8;  d[9][1]=-7;   d[9][2]=-0.2;
        d[10][0]= -8; d[10][1]=-5;  d[10][2]=-0.6;
        d[11][0]= -8; d[11][1]=-3;  d[11][2]=-1;
        d[12][0]= -7; d[12][1]=-10; d[12][2]=0.6;
        d[13][0]= -7; d[13][1]=-9;  d[13][2]=0.4;
        d[14][0]= -7; d[14][1]=-8;  d[14][2]=0.2;
        d[15][0]= -7; d[15][1]=-7;  d[15][2]=0;
        d[16][0]= -7; d[16][1]=-6;  d[16][2]=-0.2;
        d[17][0]= -7; d[17][1]=-5;  d[17][2]=-0.4;
    end
    I(a, b) <+ $table_model (V(a), V(b), d, "3S,1S");
end
endmodule
```