MIDAS User Manual

Shahriar Rabii, Louis A. Williams III,
Bernhard E. Boser, and Bruce A. Wooley

Center for Integrated Systems
Stanford University
Stanford, CA 94305

Version 3.1
October 1997

© Stanford University

MIDAS is a functional simulator for mixed digital and analog sampled-data systems. The first section of this manual describes the concepts behind MIDAS and gives some short examples of its use. The following sections provide the detailed information needed to run MIDAS.

Contents

1 Simulating with MIDAS **1**
1.1 Basic Principles 1
1.2 Running Midas 3
1.3 A Simple Example 4
1.4 The Late Output 6
1.5 Simulation Algorithm 7
1.6 Summary 9

2 Input File Format **10**
2.1 The `CONST` Section 11
2.2 The `CONTROL` Section 12
2.3 The `NETLIST` Section 13
2.4 Example: A First-Order ΣΔ Modulator 14
2.5 Example: A Second-Order ΣΔ Modulator 15
2.6 Example: A Second-Order ΣΔ Digital Noise Shaper 17
2.7 Example: Simulating Capacitor Voltage Coefficients 19

3 User Defined Models **24**
3.1 Model File Format 24
3.2 A Sample Model 27
3.3 Pin Functions 27
3.4 Model Functions 28
3.5 Compiling the Model 29
3.6 Performance Enhancement 30
3.7 Some Warnings 30

4 Writing Advanced Models **32**

References **36**

Appendices **37**

A What's New in MIDAS 3.1 **37**

B Installation **38**

C Error Messages **40**

D Model Library **43**

`abs` . . . 46
`absint` . . . 47
`add` . . . 48
`add4` . . . 49
`add4int` . . . 50
`add4intN` . . . 51
`addint` . . . 52
`addintN` . . . 53
`BHarris` . . . 54
`Blackman` . . . 56
`clkDelay` . . . 58
`clkDelayint` . . . 59
`complexIn` . . . 60
`consts` . . . 61
`cos` . . . 62
`dB` . . . 63
`decimate` . . . 64
`decimateint` . . . 65
`delay` . . . 66
`delayint` . . . 67
`dft` . . . 68
`disto` . . . 69
`distortDiff` . . . 71
`distortIter` . . . 72
`div` . . . 73
`divider` . . . 74
`divint` . . . 75
`divintN` . . . 76
`double2int` . . . 77
`double2N` . . . 78
`double2vec` . . . 79
`doubleIn` . . . 80
`exp` . . . 81

`fft` 82
`gauss` 83
`Hamming` 84
`Hanning` 86
`hexin` 88
`ifelse` 89
`ifequal` 90
`ifgreater` 91
`ifgtelse` 92
`iir` 93
`iirint` 94
`impulse` 95
`impulseint` 96
`int2double` 97
`int2N` 98
`integerIn` 99
`intermod` 100
`limiter` 102
`limiterint` 103
`linfir` 104
`linfirint` 106
`ln` 108
`matrix2vec` 109
`matrixIn` 110
`max` 111
`maxinmat` 112
`maxint` 113
`maxinvec` 114
`mean` 115
`min` 116
`mininmat` 117
`minint` 118
`mininvec` 119
`monitor` 120
`mul` 121
`mul4` 122
`mul4int` 123
`mul4intN` 124

`mulint` 125
`mulintN` 126
`mux2` 127
`neg` 128
`negint` 129
`Nuttall30` 130
`ones` 132
`pad` 133
`pdf` 134
`pow` 135
`power` 136
`powint` 137
`print` 138
`pulse` 139
`pureDelay` 140
`quant` 141
`quantint` 142
`quantizer` 143
`quantizerint` 144
`quantnlave` 145
`quantnldac` 146
`quantUniform` 147
`quantUniformint` 148
`rectangular` 149
`settle` 151
`shift` 152
`sigma1` 153
`sigma2` 154
`sin` 155
`sinc3` 156
`sinc3int` 157
`sinc4` 158
`sinc4int` 159
`sincN` 160
`sincNint` 161
`sinN` 162
`sinNzins` 164
`sinNzoh` 166

sinzins 168
sinzoh 170
skip 172
skipint 173
sqrt 174
sub 175
subint 176
subintN 177
tapDelay 178
tapDelayint 179
time 180
timeint 181
uniform 182
variance 183
vectorIn 184
zeros 185
zins 186
zinsint 187
zoh 188
zohint 189

1 Simulating with MIDAS

MIDAS is a functional simulator for sampled-data digital and analog systems. Its purpose is to close the gap between an idea for an algorithm and a circuit implementation of that algorithm. Accepting a high level functional description of a system, MIDAS frees the designer from circuit details and yet provides the flexibility to include as many circuit limitations as desired. Fundamental relationships between subsystem parameters, such as the dynamic range or bandwidth of an amplifier, and the performance of the system can quickly be examined for a variety of possible architectures. Systems with multiple sampling intervals and nonuniform sampling are supported.

Other features of MIDAS include its computation speed, which is considerably higher than that of a circuit simulator, and the availability of pre-defined models for several system building blocks. The standard models provided with MIDAS include a wide variety of linear and nonlinear system elements, as well as models for generating test signals, such as sinusoids or Gaussian noise. Several different analysis techniques are available to examine internal and external signals of the simulated system. These include transient analysis, spectral analysis, distortion analysis, and the estimation of statistical quantities such as mean, power, and probability density.

MIDAS provides two interfaces for the user. The configuration and parameters of the simulated system and the analyses to be performed are entered by means of an input file that builds on models for primitive functions such as adders, quantizers, delay elements, sinusoidal or other sources, and signal analyzers. These models can either be the standard models included with MIDAS, or user-defined models. The second user interface is the capability to add new models to MIDAS. New models are compiled directly into the main program, making their execution very fast. This also makes it easy to incorporate existing code, from program libraries for example, into new models.

1.1 Basic Principles

MIDAS deals chiefly with three entities – *models*, *pins*, and *signals*. Models are connected through their pins to the signals. A list of models forms a *netlist*, as illustrated in Figure 1. A *simulation* in MIDAS consists of multiple passes though a netlist. We will call one pass through the netlist a *simulation pass*. As MIDAS proceeds through a netlist, it executes each model. In the netlist, models can be

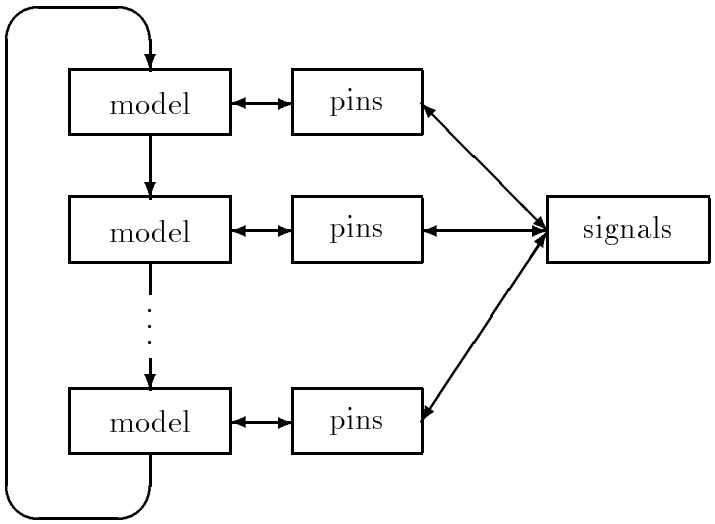

Figure 1: A block diagram of the netlist.

entered in any order; MIDAS will reorder them appropriately. Executing a model performs various operations on the signals connected to that model. Different models may be connected to the same signal, allowing information to be passed between the models.

The netlist is specified in a file called the input file. The syntax of the input file is described in Section 2. Upon startup, MIDAS parses the input file, creates appropriate data structures for the models, pins, and signals, and determines the order of execution of the models in the netlist. It then performs the simulation described in the input file's netlist.

It is often desirable to run more than one simulation over a range of system parameters using the same netlist. This can be done with one input file. Thus, we say that an input file defines an *experiment*. In an experiment, there are a set of special signals called *control signals*. A separate simulation is run for each combination of control signals.

Including control signals, there are a total of three types of signals – *constants*, *control signals*, and *net signals*. Constants are signals which do not change during the entire experiment. Control signals remain constant during each simulation, but vary between the simulations in an experiment. The values of the net signals are determined by the models in the netlist, and thus can vary during a simulation.

There is also a special class of signals called *invariant signals*. They are signals which do not change after the first simulation pass in a simulation. Constants and control signals are always invariant; net signals can be invariant under the conditions described below.

The models in a netlist are connected to signals through four different types of pins – *inputs*, *parameters*, *outputs*, and *late outputs*. Inputs can be connected to any of the three types of signals and are used to pass information to a model. Parameters can only be connected to invariant signals, and are used to pass information to a model which should not change during a simulation. Outputs and late outputs can only be connected to net signals, and are used to obtain information from a model. Note that a pin's type is part of the model's definition and as such cannot be changed by the input file.

There are also two classes of models, *algebraic* and *nonalgebraic*. Most models are algebraic. Nonalgebraic models include analysis models such as `disto` and `pdf`, time-base models such as `time`, and models containing late outputs such as `delay`. Appendix D notes all of the standard MIDAS models which are nonalgebraic. An algebraic model whose inputs are only connected to invariant signals is said to be an *invariant model*. As stated earlier, invariant signals include all constants and control signals, and some net signals. Net signals are invariant if and only if they are connected to the output of an invariant model.

The simulation algorithm used by MIDAS works as follows. For each simulation pass, each model in the netlist will be executed once, but only if one of the signals connected to one of the model's input pins has changed since the last simulation pass. Thus, the sampling period for a model is the interval between changes at one or more of its inputs.

In a netlist there can be feedback loops. Every feedback loop in the netlist must contain at least one delay. Delays are implemented using late output pins, which are discussed in Section 1.4.

MIDAS's simulation algorithm is discussed in more detail in Section 1.5.

1.2 Running Midas

The following discussion assumes an environment similar to the UNIX operating system running the C-shell. To run MIDAS, the system administrator must first install and compile the program (see Appendix B). MIDAS can then be executed in several ways, depending on where the outputs are to be sent. It may be invoked by

`midas <` *inputfile arguments*

The *inputfile* is a MIDAS netlist containing a description of the experiment to be performed including the system, the stimulus, and the desired analyses. Its format is described in Section 2. MIDAS begins its output by echoing the input file to `stderr`. If there are any errors in the netlist, error messages appear on `stderr` and the experiment is aborted. Otherwise, the simulation proceeds and the results are sent to `stdout`. Normally, `stderr` and `stdout` are connected to the screen unless specified otherwise. Therefore, when MIDAS is executed as shown above, the results will be seen on the computer monitor. The simulation output can be redirected to a file with

`midas <` *inputfile* `>` *outputfile arguments*

The *outputfile* contains the results of the simulation. The netlist echo and error messages still appear on the `stderr`, which remains connected to the screen. To redirect both `stderr` and `stdout` to the same file, the following command may be used

`midas <` *inputfile* `>&` *outputfile arguments*

Finally, to redirect the simulation results to *outputfile* and the netlist echo and error messages to *errorfile*, use

`(midas <` *inputfile* `>` *outputfile arguments* `) >&` *errorfile*

The above syntax assumes the use of the C-shell. Please consult your computer system manual for more information on redirection and file manipulation in other environments. For backward compatibility with MIDAS version 2, the output stream for the netlist echo and the error messages can easily be reconfigured to `stdout` (see Appendix B).

The *arguments* are optional. They allow the MIDAS user to incorporate command line arguments into the input file. Any occurrence of `$1` in the input file is replaced by the first argument specified on the command line, `$2` by the second argument, and so on.

1.3 A Simple Example

Before discussing the details of MIDAS, it may be helpful to first look at a simple example. The system we will simulate is the simple adder shown in Figure 2. This system is implemented with the input file shown in Figure 3. The output produced by running MIDAS on this input file is shown in Figure 4. The clock is implemented by the model `time` and the "+" block is implemented by the model `add`.

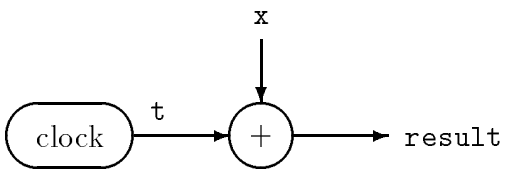

Figure 2: A simple system example.

```
CONTROL
  FOR x = [1, 2];
NETLIST
  time (kstart<-1, kstop<-3, k->t);
  add (x1<-x, x2<-t, y->result);
  print (x1<-"\nSimulation of x =", x2<-x, x3<-"\n");
  print (x1<-x, x2<-" +", x3<-t, x4<-" =", x5<-result, x6<-"\n");
END
```

Figure 3: A simple input file.

```
Simulation of x =          1
           1 +          1 =          2
           1 +          2 =          3
           1 +          3 =          4

Simulation of x =          2
           2 +          1 =          3
           2 +          2 =          4
           2 +          3 =          5
```

Figure 4: The output from a simple input file.

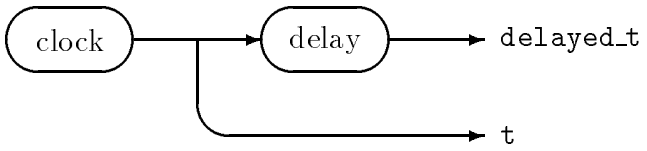

Figure 5: A simple delay example.

The input file first defines `x` as a control signal. The experiment has two simulations, one for each of the possible values of `x`. The models to be executed in each simulation are specified in the `NETLIST` section. A simulation will repetitively pass through the netlist according to a time base model such as `time`. In this example, the model `time` steps the signal `t` from 1 to 3 incrementing by one on each simulation pass. The model `add` places the sum of the current values of `x` and `t` in the signal `result`. The two instances of the `print` model send the signal values to the output file.

Note that the first `print` model is only executed once for each simulation. This illustrates an important feature of MIDAS. Models only execute if one of their inputs has changed. Since `x` only changes at the beginning of the simulations, the first `print` model only executes on the first pass through the netlist.

1.4 The Late Output

While the previous example illustrates much of the power of MIDAS, it neglects an important feature of the program; namely, the concept of the late output. We saw in the previous example that MIDAS will make multiple passes through a list of models. In that example, when the `add` model was executed, the output was immediately placed in the signal `result`. In MIDAS, there is a special class of output pins called late outputs for which this is not the case. When a model has a late output that has changed, the change does not take effect until the beginning of the next simulation pass. In this manner, delays can be simulated.

The late output is illustrated with a simple system whose block diagram is shown in Figure 5. The output file from the simulation of this system, including the netlist echo, is shown in Figure 6. You have already seen the models `time` and `print` in the previous example. Here we introduce the `delay` model. This model copies the value of its input `t` to its output `delayed_t`. But the output of `delay` is a late output pin, so the value change does not take effect until the next pass through the netlist. As is

```
#  ************************************************************
#                            MIDAS (TM)
#                           Version  3.1
#                       Copyright  (C) 1989
#  The Board of Trustees of the Leland Stanford Junior University
#                       All Rights Reserved
#  ************************************************************
#
# NETLIST
#   time (kstart<-1, kstop<-4, k->t);
#   delay (x1<-t, y->delayed_t);
#   print (x1<-"t:", x2<-t, x3<-",   delayed_t:", x4<-delayed_t,
#          x5<-"\n");
# END
#
t:           1,   delayed_t:           0
t:           2,   delayed_t:           1
t:           3,   delayed_t:           2
t:           4,   delayed_t:           3
```

Figure 6: An illustration of the late output.

apparent in the output from the `print` model, this has the effect of a one clock cycle delay.

1.5 Simulation Algorithm

Having seen the basics of MIDAS's operation, we are now ready to discuss the simulation algorithm in detail. Before an experiment begins, MIDAS reorders the models so that they can be executed in order. Then, MIDAS finds all of the invariant models and puts them in a separate list. These initial steps are called the parsing phase of MIDAS's operation.

After the parsing is complete, MIDAS begins its simulation algorithm. This algorithm is summarized in Figure 7.

Before discussing the algorithm itself, however, a brief digression into model operation is warranted. Each model has up to three operations or functions that it can perform. These three functions are `Initialize`, `Execute`, and `LastExecute`.

For each combination of control signals:

    For each invariant model:
        Call the `Initialize` function.
        Call the `Execute` function.
    For each model in the netlist:
        Call the `Initialize` function.
    Loop (until the simulation is done):
        Update the late outputs.
        For each model in the netlist:
            If an input pin has changed, call the `Execute` function.
    For each model in the netlist:
        Call the `LastExecute` function.

Repeat until control signals are exhausted.

Figure 7: The MIDAS simulation algorithm.

The `Initialize` function is used to initialize the model's internal state. The `Execute` function is the essence of most models. This is where most operations on signals are performed and where most outputs are generated. The `LastExecute` function provides the mechanism through which models perform operations at the end of a simulation. In most models, this function just calls the `Execute` function one last time if one of the model's input signals has changed. Models for which `LastExecute` is more complicated include most of the analysis models, where information about the system is accumulated over the course of the simulation and analyzed at the end of the simulation. (See, for example, the description of the `pdf` model at the end of this manual.)

We now return to the algorithm in Figure 7. For each combination of control variables, a simulation is performed. Each simulation proceeds as follows. First, all of the invariant models are initialized and executed. Next, all of the remaining models are initialized. Then, the main simulation loop is entered. MIDAS remains in this loop as long as specified by some time base type of function (such as `time`).

As stated in the description of the late outputs, their value changes do not take effect until the beginning of the next simulation pass. Thus, the first operation in the simulation loop is the updating of the late outputs. Note that when the simulation loop is entered for the first time, this operation has the effect of initializing the late outputs.

After the late outputs are updated, for each model in the netlist which has an input pin that has changed since the last simulation pass, the model's `Execute` function is called. Note that when a control signal changes between simulations, it is considered "changed" on the first simulation pass. Also, if all of the model's input pins are connected to constants (or unconnected), then the model's `Execute` function will be called on the first simulation pass of the first simulation in the experiment.

Finally, when the main simulation loop is exited, the `LastExecute` function is executed for each of the non-invariant models. This is where most analysis functions are performed. As noted earlier, in many models, the `LastExecute` function simply calls the `Execute` function if any of the model's inputs have changed. However, because of the way in which the time base models `time` and `hexin` are defined, the `LastExecute` loop is very different from a normal simulation loop. In the `LastExecute` loop, the outputs of the time base models do not change, so there is no clock. Furthermore, the late outputs are not updated. Thus, the only signals which normally change in the `LastExecute` loop are the outputs of the analysis models and any outputs of models which are connected to the analysis outputs.

1.6 Summary

In this section, we have presented the basic operation of MIDAS. The following sections contain the details of the input file syntax, some examples, and the procedures for writing user-defined models. The appendices contain installation information, the error messages of MIDAS, and descriptions of all of the standard models supplied with MIDAS.

2 Input File Format

The system, its stimuli, and the desired analyses are specified in the input file. The input file has the following basic structure

```
CONST
    constant definitions
CONTROL
    control ranges
NETLIST
    model statements
END
```

The formats for the *constant definitions*, *control ranges*, and *model statements* are given below. The three sections must appear in the order indicated. The `CONST` and `CONTROL` sections are optional; the `NETLIST` section is required.

In an input file, white space (blanks, tabs, and newlines) is usually ignored, but sometimes serves as a separator between names. Thus extra white space never hurts, and the lack of white space rarely causes trouble.

Comments are allowed in input files. Comments are enclosed between `(#` and `#)`, and may be nested. An alternate form for comments begins with an `#` symbol and continues to the end of the line. (This is the only instance in which a newline is not simply treated as a space.)

Also allowed anywhere in the input file are file include statements, which have the following format

`INCLUDE <`*filename*`>;`

MIDAS will read the contents of the file *filename* as if they were part of the input file. When the include file has been completely read, MIDAS will return to reading the input file. The include file can contain its own include statements. Up to ten levels of include files can be nested. *filename* may include a tilde character, $\sim$, as a shorthand to denote the home directory of a user.

As discussed in Section 1, one of the important entities in the input file is the signal. Signals can appear in all three sections of the input file. Thus, there are three basic types of signals – constants, control signals, and net signals. The signal type is determined by the section in which the signal first appears. Each signal has a value, and there are seven different types of values – double, integer, complex, vector, matrix,

string, and stream. A double type is a double precision real number. Integers are of the type `long`, the largest integer allowed by the computer. Integers are typically generated and used by models that are used in simulating digital systems. Only net signals may be integers. A complex type is a complex number represented by real and imaginary parts, each of which are double precision real numbers. A vector type is a one dimensional array of double precision real numbers. A matrix type is a two dimensional array of double precision real numbers. A string type is a sequence of characters. Finally, a stream type is an I/O specifier, such as a file or a terminal. The signal value type is determined where the signal is defined.

2.1 The `CONST` Section

The `CONST` section of the input file consists of *constant definitions*, which have the following format

constantname `=` *constant expression* `;`

The *constantname* defines the constant's name, and can be any sequence of letters and digits that begins with a letter. The underscore character "`_`" is treated as a letter by MIDAS. Upper and lower case letters are distinct.

A *constant expression* can be either a number, a vector, a string, a scalar expression, a data file or a previously defined constant. Numbers are entered in the obvious manner. Exponential notation (e.g., `1.0e-5`) is supported. Vectors are delimited by "`[`" and "`]`". Each vector element can be either a number or a scalar expression. Vector elements are separated by commas. Strings are delimited by quotation marks (`"`). All of the standard C escape sequences are recognized. Scalar expressions are any combinations of numbers, previously defined constants, and the four basic algebraic operators, `+`, `-`, `*`, and `/`. Parentheses can be used to group expressions.

A data file constant is somewhat special in that when it is defined, the data file is opened. There are two formats for a data file constant. They are

`<`*filename* `,` *type_of_io*`>`
`<`*filename*`>`

The *filename* is any legal filename, and the *type_of_io* is one of: `read`, `write`, `append`, `r+`, `w+`, or `a+`. If *type_of_io* is omitted, the default is `read`. The "`+`" modes are not normally used, but they are needed for some models which perform more advanced file manipulation. The non-"`+`" modes can be abbreviated with their first letter.

Some examples of valid constant definitions are

```
c1 = 0.5;
bruce = -1234;
a_vector = [ 1, c1, 3, 2*2, 2+3 ];
aString="Hello MIDAS\n";
c2 = (-4+7)/3.1;
file = <mydata>;
ofile = <calcs.dat, w>;
```

Integer, complex, and matrix constants are not implemented. However, these and other data types may be stored in files and read in with the appropriate models.

2.2 The `CONTROL` Section

The `CONTROL` section of the input file consists of *control ranges* that define the control signals. Control signals have some of the properties of both constants and net signals. For a given simulation, the control signals act as constants. But, unlike constants, control signals step through a range of values, and a new simulation is run for each combination of control values. In the first simulation pass of each simulation, control signals act as if they were net signals whose value has changed. Thus, any model with input pins connected to a control signal will execute during the first simulation pass of a simulation.

Control ranges have the following possible formats

`FOR` *controlname* `=` *start* `TO` *stop* `STEPS` *no_of_steps*`;`
`FOR` *controlname* `=` *start* `TO` *stop* `LOG STEPS` *no_of_steps*`;`
`FOR` *controlname* `=` *vector literal*`;`

The formats of the *controlname* and *vector literal* are the same as the constant name and vector formats in the `CONST` section. *Start*, *stop*, and *no_of_steps* are all *constant expressions*. The control variable will step through the range from *start* to *stop*, dividing that range into *no_of_steps* values of step size

$$\frac{stop - start}{no_of_steps - 1}$$

If the `LOG` form is used, the range will be divided logarithmically. In the vector form, the control variable will step through each element of the vector.

MIDAS imposes no limit on the number of control variables which can be defined in an input file. A complete simulation of the `NETLIST` will be done for every possible

combination of control variables. This is done by treating the `FOR` statements as being nested, with the last `FOR` statement varying the fastest.

2.3 The `NETLIST` Section

The `NETLIST` section is the heart of the input file. It defines the simulation which is to be carried out. This section contains a series of *model statements* which connect signals and models together. The syntax for a model statement is

model`(`*pin connections*`);`

The *model* is the name of any standard or user defined model. The *pin connections* are made up of any number of pin connections separated by commas. Each pin connection has the form

pinname connector signalname

The possible *pinnames* for the standard models are given in the model descriptions (see Appendix D). The *connector* can be `->` or `<-`. The *signalname* has the same format as a constant name in the `CONST` section, and is the name of the signal to which the model is being connected. Note that signal names are stored separately from the pin and model names, so signal names which are the same as model or pin names are legal and will not confuse MIDAS. The pin connections can appear in any order within the model statement. Furthermore, all of the pin names that are defined for a model do not need to be connected to a signal. Unconnected inputs and parameters will be connected to a default constant defined in the model. Values of the default constants are given in the model descriptions (see Appendix D). Unconnected outputs and late outputs will be connected to dummy signals.

There are four types of model pins – input, parameter, output, and late output pins. Input and parameter pins are connected with the connector `<-`. Input pins can be connected to any type of signal or constant expression. Parameter pins can be connected to any invariant signal or constant expression. Output and late-output pins can only be connected to a net signal using the connector `->`. Signals that have not appeared in the `CONST` or `CONTROL` sections are defined to be net signals. Net signals must be driven by exactly one output or late-output pin. However, any number of inputs and parameters can be connected to a signal.

Model statements may appear in any order. MIDAS will rearrange them so they can be executed in order. For each pass through the `NETLIST`, MIDAS will only execute the model statements for which a signal connected to an input pin has changed value.

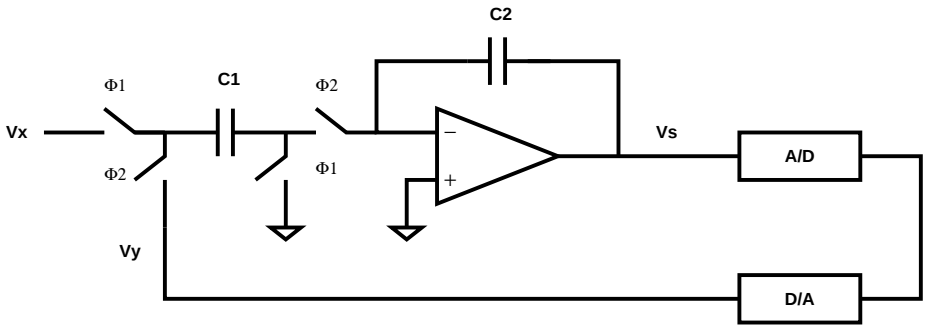

Figure 8: A first-order ΣΔ modulator.

2.4 Example: A First-Order ΣΔ Modulator

This section shows how to generate a MIDAS description for an implementation of a first-order ΣΔ modulator. An understanding of switched-capacitor circuits at the level of [1] is assumed. Figure 8 shows the simplified circuit diagram of a first-order ΣΔ modulator [2]. The circuit is composed of an operational amplifier, switches, capacitors, a 1-bit comparator, and a 1-bit DAC. The amplifier, switches, and capacitors constitute a switched-capacitor integrator. Denoting clock phase Φ1 with index `n` and clock phase Φ2 with index `n` $+\frac{1}{2}$ we can write the following difference equation for the integrator

$$V_s[n+1] = V_s[n] + \frac{C_1}{C_2}(V_x[n] - V_y[n+\frac{1}{2}])$$

Since the quantizer is clocked on Φ1 and holds its previous value during Φ2, $V_y[n+\frac{1}{2}] = V_y[n]$. Therefore, the integrator difference equation can be rewritten as

$$V_s[n+1] = V_s[n] + \frac{C_1}{C_2}(V_x[n] - V_y[n])$$

The integration function is performed by embedding a delay element in a feedback loop. The 1-bit A/D converter (comparator) produces a digital `1` if the analog input is greater than or equal to 0 and a digital `0` if the analog input is less than 0. Furthermore, since the output of the 1-bit D/A converter simply maps the digital output of the 1-bit A/D converter to analog voltages, the information content of these signals is identical. It is convenient to combine the A/D and D/A blocks into a single quantizer block and perform signal analysis on the analog output of the quantizer.

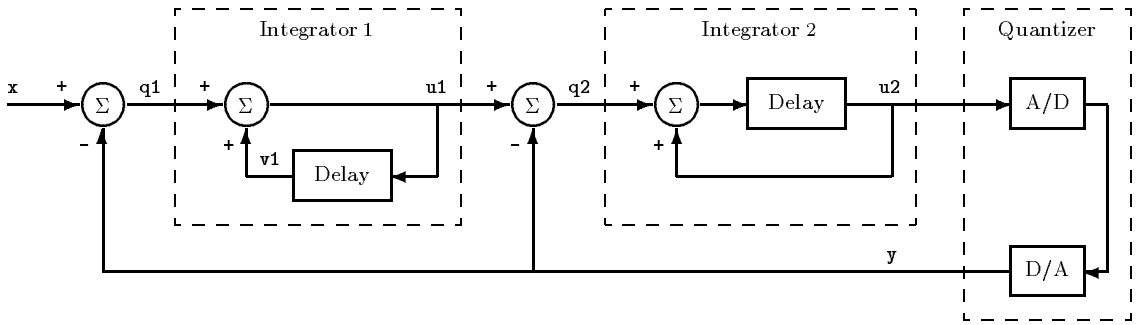

Figure 9: Block diagram of a second-order ΣΔ modulator.

Assuming a capacitor ratio of 1/2, we can now describe the above system with the following netlist fragment

```
(# first-order sigma-delta modulator #)
sub (x1<-x, x2<-y, y->n1);        (# subtract feedback from input #)
mul (x1<-n1, x2<-0.5, y->n2);     (# integrator gain #)
add (x1<-n2, x2<-s, y->n3);       (# integrator described in 2 lines #)
delay (x1<-n3, y->s);
quant (x<-s, y->y);               (# quantizer #)
```

Note that the output of the `quant` model is $\pm\frac{1}{2}$ by default. The above netlist fragment can be made shorter and the MIDAS execution time can be reduced by taking advantage of some additional features of the `delay` model

```
(# first-order sigma-delta modulator #)
sub (x1<-x, x2<-y, y->n);         (# subtract feedback from input #)
delay (x1<-n, x2<-s, y->s,
       c1<-0.5);                  (# integrator with gain of 0.5 #)
quant (x<-s, y->y);               (# quantizer #)
```

The reader is encouraged to confirm that these two netlist fragments do, in fact, implement the same system. The next sections provide more elaborate examples.

2.5 Example: A Second-Order ΣΔ Modulator

Figure 9 shows the block diagram of the core of an A/D converter, a second-order ΣΔ modulator [3], [4]. The input file is given in Figure 10. In the following discussion, the terminology introduced in Section 1.1 will be used.

The `CONST` section of the input file declares a number of important constants. A sampling frequency of 1 MHz and a test signal frequency of 109 Hz are defined here. The signal frequency is relatively low so that the harmonics of the signal fall

in-band and are not removed by the decimation filter. The 0-dB reference level of the input source is set equal to 0.5 with `amplitude`, which is also equal to the magnitude of the feedback reference voltages of the quantizer. Therefore, a 0-dB input signal corresponds to the maximum theoretical input level of the modulator before severe clipping takes place. Modulators typically experience overload a few dB below this level. The `cycles` constant is used to set the duration of each simulation in the experiment to 6 cycles of the input signal. Finally, `h48` is a vector that contains half of the coefficients of a symmetric, linear phase, FIR filter used in the decimation of the modulator output.

The `CONTROL` section determines that the experiment will be carried out for all combinations of two signals, each of which can take on one of four values. The `M` signal controls the oversampling ratio, or, equivalently, the decimation ratio, and the `gain` signal controls the input amplitude. Thus, there will be 16 distinct simulations in the experiment.

The `NETLIST` section begins with a signal source created by the `time` and `sin` models. The time increments are equal to the sampling period and the stop time is selected to accomodate 6 cycles of the input signal. Note that since `cycles`, `fx`, and `fs` are constants, they can be used in arithmetic expressions and do not require explicit divide models to set the parameters of `time`. The amplitude of the sinewave is set through the `A` and `gain` pins of the `sin` model.

The next few lines describe the second-order modulator. It should be verified by the reader that this description is an accurate representation of the block diagram shown in Figure 9. Next, the output of the modulator is decimated by a factor of $M/2$ in a comb filter and a factor of 2 in an FIR filter. Since `M` is not a constant, an explicit `div` model is need to divide it by 2. However, since `M` is an invariant signal, this computation is performed only once per simulation. Therefore, there is little performance penalty due to the use of the `div` model.

The output of the decimation filter is analysed by the `disto` model. This model implements an LMS algorithm that computes various properties of the input sequence, including the signal-to-noise ratio and the mean squared error. For more information about this and other models, see Appendix D. Finally, the results of the simulations are printed to `stdout` using the `print` model. Note that in the first call to `print`, whenever either of its input signals change, all inputs are printed. In the second call to `print`, since the `trigger` pin is used, the inputs are printed only when `mse` changes. This insures that the output is generated only after the `disto` model has completed its operation and not when a control signal changes.

The output of this experiment shows the relationship between oversampling ratio, signal-to-quantization noise, and mean squared error as the input signal power is varied.

2.6 Example: A Second-Order $\Sigma\Delta$ Digital Noise Shaper

The MIDAS input file for a linear interpolator and digital noise shaper is given in Figure 11. This system implements the core of a $\Sigma\Delta$ D/A converter. Detailed information about this architecture can be found in [5].

The `CONST` section declares a Nyquist sampling frequency of 44.1 kHz and an oversampling ratio of 176. The test signal frequency is 3.1 kHz. The `cycles` constant is used to set the duration of each simulation in the experiment to 10 cycles of the input signal. `bits` specifies the resolution of the signal source as well as some of the arithmetic blocks in the netlist. `harm` is the number of harmonics of the input signal to search for in the noise shaper output. Finally, the `INCLUDE` statement inserts a file that defines a vector, `h48`, containing the coefficients of a reconstruction filter. The filter may be identical to that used in the FIR decimator of Section 2.5.

The 0-dB reference level of the input source is set equal to 32768 with `amp`. This number is equal to 2^{15} and represents the peak amplitude of a 16-bit digital sinusoid. It is also equal to the magnitude of the feedback reference values of the quantizer. The `CONTROL` section steps up the amplitude of the input source in 1-dB increments from -95 dB to 0 dB for each simulation. Thus, the experiment described by this input file consists of 96 simulations.

The `NETLIST` section begins with a signal source created by the `time` and `sinNzoh` models. The time increments are equal to the sampling period and the stop time is selected to accomodate 10 cycles of the input signal. The amplitude of the sinewave is set through the `A` and `gain` pins. The output of the `sinNzoh` model is samples of a sinewave with 15 zeros inserted between them. Note that for low values of `gain`, an amplitude of less than 1 LSB may be specified. In this situation, the output of `sinNzoh` is rounded up and oscillates with an amplitude of 1 LSB. In this experiment, when `gain` drops below -90-dB, the output of the source remains unchanged.

The zero-order held input signal is fed to a linear interpolator to generate the intermediate values. A detailed description of this circuit can be found in [6]. The next set of commands define the second-order noise shaper. Next, the output of the noise shaper is decimated by a factor of $M/2$ in a comb filter and a factor of 2 in an FIR filter. The output of the decimation filter is analysed by the `disto` model.

Finally, the results of the simulations are printed to `stdout` using the `print` model. In the second call to `print`, since the `trigger` pin is used, the inputs are printed only

```
CONST

  fs = 1.0e6;           (# sampling rate                        #)
  fx = 109;             (# input signal frequency               #)
  amplitude = 0.5;      (# 0-dB amplitude                       #)
  cycles = 6;           (# cycles of fx in simulation           #)
  h48 = [ -3.953e-3, -5.929e-3,  5.929e-3,  3.953e-3, -5.929e-3,
          -7.905e-3,  7.905e-3,  9.881e-3, -1.186e-2, -1.383e-2,
           1.383e-2,  1.779e-3, -1.976e-2, -2.372e-2,  2.569e-2,
           3.162e-2, -3.557e-2, -4.545e-2,  4.941e-2,  7.115e-2,
          -7.510e-2, -1.324e-1,  1.403e-1,  5.040e-1 ];
                        (# coefficients of decimation filter    #)
CONTROL

  FOR M = [512, 256, 128, 64];
  FOR gain = [-20, -5, -2, 0];

NETLIST

  (# source #)
  time (k->kT, kstop<-(cycles/fx), kstep<-(1/fs));
  sin (t<-kT, y->x, f<-fx, A<-amplitude, gain<-gain);

  (# second-order sigma-delta modulator #)
  sub (x1<-x, x2<-y, y->q1);        (# first summing node #)
  add (x1<-q1, x2<-v1, y->u1);      (# first integrator #)
  delay (x1<-u1, y->v1);
  sub (x1<-u1, x2<-y, y->q2);       (# second summing node #)
  delay (x1<-q2, x2<-u2, y->u2);    (# second integrator #)
  quant (x<-u2, y->y);              (# quantizer #)

  (# decimation filter #)
  div (x1<-M, x2<-2, y->M_2);
  sinc3 (x<-y, y->y1, M<-M_2);
  linfir (x<-y1, y->Y, h<-h48, M<-2);

  (# analysis and output #)
  div (x1<-fs, x2<-M, y->fn);
  div (x1<-fx, x2<-fn, y->fx_fn);
  disto (x<-Y, fxT<-fx_fn, snr->snr, mse->mse);
  print (x1<-"\nfn: ", x2<-fn, x3<-", M: ", x4<-M, x5<-"\n",
         format<-".5g");
  print (x1<-gain, x2<-snr, x3<-mse, x4<-"\n", trigger<-mse);

END
```

Figure 10: Input file for a second-order modulator.

when `sndr` changes. This insures that the output is generated only after the `disto` model has completed its operation and not when `gain` changes. The output of this experiment plots the signal-to-quantization noise plus distortion ratio of the system as the input amplitude is swept.

2.7 Example: Simulating Capacitor Voltage Coefficients

As an illustration of simulating circuit nonidealities, this section examines the modelling of capacitor voltage coefficients in a $\Sigma\Delta$ modulator. The impact of this nonlinearity on the performance of a modulator varies considerably with the specific circuit implementation. The following discussion focusses only on the first integrator and assumes the integrator topology shown in Figure 8. In this Figure, C_1 is the sampling capacitor and C_2 is the integration capacitor.

Assume the capacitors can be described by

$$C(V) = C_0 \cdot (1 + \alpha_1 V + \alpha_2 V^2)$$

C_0 is the nominal capacitance, α_1 is the capacitor linear voltage coefficient, and α_2 is the capacitor quadratic voltage coefficient. It can be shown that the integrator difference equation is

$$V_{out}[n+1] = V_{ideal} + V_{distC1} + V_{distC2}$$

The output voltage is equal to the ideal output plus an error due to the sampling capacitor voltage coefficient and an error due to the integration capacitor voltage coefficient. The ideal output is

$$V_{ideal} = V_{out}[n] + \frac{C_{01}}{C_{02}}(V_x[n] - V_y[n])$$

C_{01} and C_{02} are the nominal capacitances of the sampling and integration capacitors, respectively. The error due to the sampling capacitor is

$$V_{distC1} = \frac{C_{01}}{C_{02}}(\frac{\alpha_1}{2}(V_x^2[n] - V_y^2[n]) + \frac{\alpha_2}{3}(V_x^3[n] - V_y^3[n]))$$

And, the error due to the feedback capacitor is

$$V_{distC2} = \frac{\alpha_1}{2}({V_{out}}^2[n+1] - {V_{out}}^2[n]) + \frac{\alpha_2}{3}({V_{out}}^3[n+1] - {V_{out}}^3[n]))$$

Note that the error due to the feedback capacitor makes the integrator difference equation a cubic polynomial in $V_{out}[n+1]$. To simulate an integrator in MIDAS with

```
CONST
  fn = 44.1e3;          (# Nyquist rate = 44.1 kHz #)
  M  = 176;             (# total oversampling ratio, M = 11*16 #)
  fs = fn*M;            (# sampling rate #)
  fx = 3.1e3;           (# input signal frequency #)
  Mzoh = 16;            (# zero-order hold samples #)
  cycles = 10;          (# cycles of input sine in simulation #)
  bits = 16;            (# number of bits in input signal #)
  harm = 5;             (# number harmonics to test for in disto #)
  amp = 32768;          (# amplitude of input signal #)
  INCLUDE <filter2.i>;  (# filter coefficients #)

CONTROL
  FOR gain = -95 TO 0 STEPS 96;

NETLIST
  (# source: a 176x oversampled input that has a 16x zero-order hold #)
  time (k->kT, kstop<-(cycles/fx), kstep<-(1/fs));
  sinNzoh (t<-kT, y->zoh, A<-amp, gain<-gain, f<-fx, n<-bits, M<-Mzoh);

  (# Linear interpolation #)
  subintN (x1<-zoh, x2<-itpl, n<-bits, y->diff, clip<-1);
  delayint (x<-diff, y->diffd);
  decimateint (x<-diffd, M<-Mzoh, y->diffdecim);
  addintN (x1<-diffdecim, x2<-itpld, n<-(bits+4), y->itpldiv, clip<-1);
  delayint (x<-itpldiv, y->itpld);
  double2int (x<-bits, y->bitsint);
  divintN (x1<-itpldiv, x2<-bitsint, n<-bits, y->itpl, clip<-1);

  (# 2nd order sigma-delta #)
  subintN (x1<-itpl, x2<-y, y->s1, n<-(bits+1), clip<-1); (#1st summer#)
  addintN (x1<-s1, x2<-d1, y->d1d, n<-(bits+2), clip<-1);
  delayint (x<-d1d, y->d1);
  subintN (x1<-d2, x2<-y2, y->s2, n<-(bits+2), clip<-1);  (#2nd summer#)
  addintN (x1<-d1, x2<-s2, y->d2d, n<-(bits+3), clip<-1);
  delayint (x<-d2d, y->d2);
  quantint (x<-d2, y->y, yp<-amp, ym<-(-amp));
  mulint (x1<-y, x2<-2, y->y2);

  (# filtering, analysis and output #)
  sinc3int (x<-y, y->ysinc, M<-M/2);
  linfir (x<-ysinc, y->Y, h<-h64, M<-2, skip<-70);
  disto (x<-Y, fxT<-(fx*M)/fs, harmonics<-harm, tsnr->sndr);
  print (x1<-"\"DAC SNDR\n");
  print (x1<-gain, x2<-sndr, x3<-"\n", trigger<-sndr);
END
```

Figure 11: Input file for a second-order digital noise shaper.

nonlinear sampling capacitors, `mul`, `add`, and `sub` models can be used. To facilitate this operation, the `distortDiff` model may also be used. Simulating the error due to the feedback capacitor is more involved because a root of a third-order polynomial must be found. The `distortIter` model solves the cubic by an iterative method. The model works well as long as the voltage coefficients are small. In a $\Sigma\Delta$ modulator, the error due to the feedback capacitors is greatly attenduated in the baseband by noiseshaping. Therefore, this term is often omitted to speed up simulations.

The netlist of Figure 12 simulates a second-order modulator whose first integrator has distortion due to capacitor voltage coefficients. The capacitor technology is assumed to provide a linear voltage coefficient of 2000 ppm/V and a quadratic voltage coefficient of 600 ppm/V^2. Since it is assumed that the actual implementation will use fully differential circuits the simulation uses a linear voltage coefficient of 40 ppm/V to account for the attenuation of even order nonlinearities. The simulation is carried out for 200 cycles of a -4-dB input signal. The modulator output is then decimated and windowed by a window with very low sidelobes. The `fft` model estimates the spectrum of the output, which is illustrated in Figure 13.

```
CONST
  fs = 6.4e6;                   (# sampling rate = 6.4 MHz   #)
  M  = 128;                     (# oversampling ratio = 128  #)
  fn = fs/M;                    (# Nyquist rate  = 50 kHz    #)
  fx = 6013;                    (# input signal frequency    #)
  gain = -4;                    (# gain of input signal      #)
  cycles = 200;                 (# cycles of input sine in simulation #)
  delta = 2.0;                  (# quantizer feedback level: +/- 2.0 #)
  harm = 7;                     (# number harmonics to find in disto #)
  intgain = 0.5;                (# integrator gains #)
  a1 = 40e-6;                   (# linear voltage coefficient #)
  a2 = 600e-6;                  (# quadratic voltage coefficient #)
  INCLUDE <filter2.i>;          (# filter coefficients #)

NETLIST
  (# source #)
  time (k->kT, kstop<-(cycles/fx), kstep<-(1/fs));
  sin (t<-kT, y->x, f<-fx, gain<-gain, A<-delta);

  (# second-order modulator #)
  (# distort the input due to sampling caps #)
  sub (x1<-x, x2<-y, y->x_y);
  distortDiff (x1<-x, x2<-y, alpha1<-a1/2, a2<-a2/3, y->x_y_dist);
  add4 (x1<-x_y, x2<-x_y_dist, y->id, a1<-intgain, a2<-intgain);

  (# estimate the distortion due to feedback caps and integrate #)
  distortIter (xin<-id, xfb<-d1, alpha1<-a1/2, alpha2<-a2/3, y->d1d);
  pureDelay (x1<-d1d, y->d1);

  (# second integrator #)
  add4 (x1<-d1,  x2<-y, y->s2,
        a1<-intgain, a2<-(-intgain));          (# second summing node #)
  delay (x1<-s2, x2<-d2, y->d2);               (# second integrator #)
  quant (x<-d2, y->y, yp<-delta, ym<-(-delta)); (# quantizer #)

  (# decimation filter #)
  sinc4 (x<-y, y->ysinc, M<-M/2);
  linfir (x<-ysinc, y->Y, h<-h64, M<-2, skip<-70);

  (# analysis and output #)
  Nuttall30 (x<-Y, w->windowed, reference<-delta);
  fft (x<-windowed, fs<-fn, S->S);
  disto (x<-y, fxT<-(fx/fs), harmonics<-harm, tsnr->tsnr);
  print (x1<-"\n\"Non-lin caps\n");
  print (x1<-S);
END
```

Figure 12: Input file for a modulator with capacitor voltage coefficients.

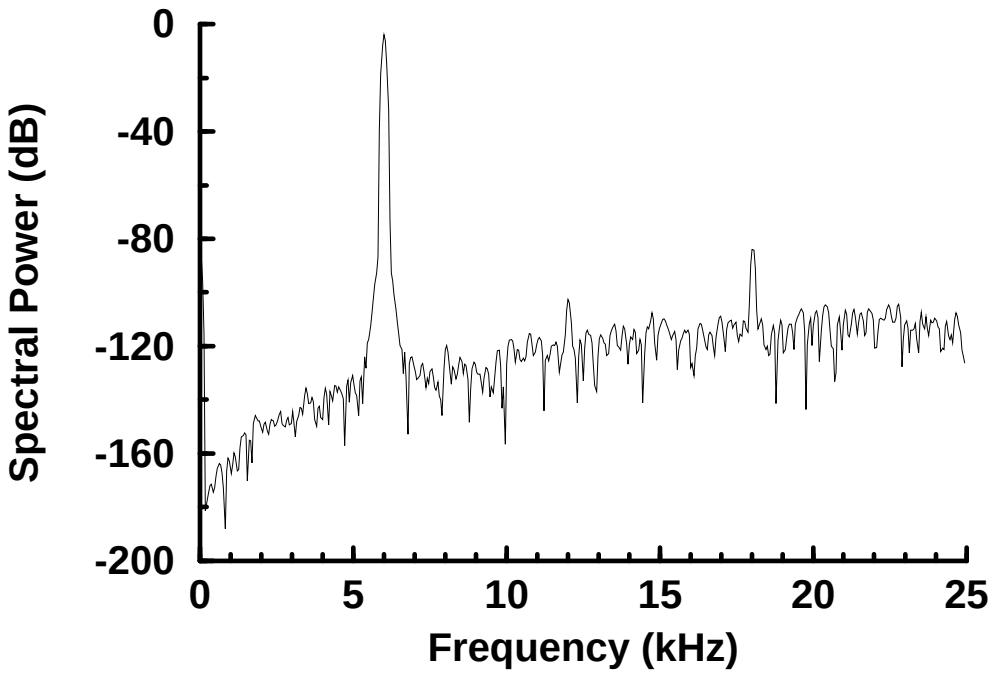

Figure 13: Baseband spectrum of modulator with nonlinear capacitors.

3 User Defined Models

This section describes how to add user-defined models to MIDAS. Models are used to add greater functionality to MIDAS. A model can define a new arithmetic block, additional file I/O, new data analysis procedures, ... Since MIDAS has no structure which can be used to combine models into a subcircuit, a new model can be defined to replace common combinations of other models. Models can also be written to access other programs.

To create a new MIDAS model, the model code must be written in the format described below. Next, MIDAS must be recompiled to incorporate the new model into the simulator. The compilation procedure is outlined in Section 3.5 and in Appendix B. Substantial effort has gone to making this process as painless as possible. While MIDAS is written in the language C++ [7], most models can be written with only a rudimentary knowledge of the language C, a subset of C++.

3.1 Model File Format

Model files, like input files, are simple text files. Model file names must have the suffix "`.m`". A model file has the following format

c-code
`MODEL` *modelname*
    definitions
`INITIALIZE`
    model-code
`EXECUTE`
    model-code
`LASTEXECUTE`
    model-code
`END`
c-code

The details of this format are described below. As in the input file, white spaces (blanks, tabs, and newlines) serve as separators. They must be used to separate alpha-numeric items from each other, but are otherwise optional.

Comments are allowed throughout the model file. As in input files, comments are enclosed between `(#` and `#)` and may be nested. However, the single line form of comments allowed in input files is not legal in a model file.[1]

The *c-code* that appears at the beginning and end of a model file can be any series of C++ statements, with the following restrictions. No variable or pin defined in the `MODEL` section can be used and C++ style comments (both the `/* */` and `//` forms) are strongly discouraged. While these comments are, strictly speaking, legal, and most of the time will not cause any trouble, they have the potential to confuse the model file processor in ways which may be difficult to decipher.

The *modelname* is the identifier you will use to refer to this model in your input files. It can be any legal identifier, i.e., a sequence of letters and digits beginning with a letter, with the sole restriction being that all model names must be distinct. This restriction includes the names of the standard models, since once MIDAS is compiled, it cannot differentiate between user defined and standard models.

The *definitions* can be either ordinary C++ variable definitions, or model pin definitions. The ordinary C++ definitions define variables which are internal to the model and can be accessed from any of the *model-code* sections. Model pin definitions create a model pin and associate a pin type and a value type with that pin. They have the following format

value-type pin-type pinnames;

The *value-type* defines the signal value type to which the pin must be connected. As mentioned earlier, the possible signal value types are `double`, `integer`, `complex`, `vector`, `matrix`, `string`, and `stream`. The *pin-type* describes how the signal is to be used. The pin types are `input`, `parameter`, `output`, and `lateoutput`. The *pinnames* are comma separated pin names. Pins whose value type is `double` or `integer` can optionally be initialized. The format for an initialized pin name is

pinname `=` *constant*

where a *constant* is any real number, integer, or C++ constant defined in the *c-code* at the beginning of the model file.

The *model-code* can contain any C++ statement that could appear inside a C++ function definition, i.e., anything except for additional function definitions. The operations a model will perform on the signals connected to it are defined in the *model-code*. By treating the model pins as if they were ordinary C++ variables, the model writer can take the value of a model pin or assign a value to a model pin, with

[1] The elimination of the single line form of comment was needed in order that C++ preprocessor directives such as "`#include`" not be treated as comments.

the exception being that input and parameter pins cannot have values assigned to them.[2] When the model is actually executed by MIDAS, instances of the model pins in the model definition will be replaced by the signals connected to those pins. Also available in the *model-code* are some special purpose functions which are described in Sections 3.3 and 3.4.

The `INITIALIZE`, `EXECUTE`, and `LASTEXECUTE` sections are converted by the model file compiler into C++ functions that are executed by MIDAS at various times. `INITIALIZE` is executed once at the beginning of each simulation, which is once for each combination of control variables. It is executed before any of the other sections. Because of this, the only pins whose value has any meaning in an initialize section are the parameter pins. Their value is valid because they are required to be connected to invariant signals. The `INITIALIZE` section is optional. `EXECUTE` is executed once every simulation pass (except for the last simulation pass), but only if one of the signals connected to the model's input pins has changed value. The `EXECUTE` section is mandatory. `LASTEXECUTE` is executed on the last simulation pass, and is useful in models which perform analyses on the simulation as a whole. This section is optional. If it is omitted, MIDAS will use a default function which executes the code in the `EXECUTE` section one last time if any of the signals connected to the model's input pins has changed value. However, as noted in Section 1.5, the only signals that normally change during the `LastExecute` loop are those connected to the outputs of analysis models.

An exception to the above execution rules occurs when the inputs to an algebraic model are all invariant, making the model invariant. (Invariant models were described in Section 1.) These models will have their `INITIALIZE` and `EXECUTE` sections executed at the beginning of each simulation before any of the non-invariant models are initialized. This has the effect of initializing all of the invariant signals.

The characteristics which make a model algebraic can now be made explicit. A model is algebraic unless (1) one of its pins is a late output, or (2) the model function `Enable` (see Section 3.4) is used in any of the *model-code* sections, or (3) there is a `LASTEXECUTE` section in the model file. When a model file is compiled, the model file translator will determine whether or not a model is algebraic.

[2] To clarify what is meant by "assign", consider the following statement.

```
y = x
```

Because it contains an equal sign, the statement is called an assignment statement. The value of `x` is "assigned to" `y`, which means that the value stored in `y` is set equal to the value stored in `x`. If `x` is a model pin, the value taken will actually be the value of the signal connected to pin `x`. If `y` is a model pin, the value of `x` will actually be assigned to the signal connected to pin `y`.

```
(# File: sin.m #)

#include <math.h>

MODEL sin

    double input t;
    double output y;
    double parameter A = 1.0, gain = 0.0, f = 1.0;
    double a, omega;

INITIALIZE

    a = A * pow(10.0, gain/20.0);
    omega = 2.0 * M_PI * f;

EXECUTE

    y = a * sin(omega * t);

END
```

Figure 14: Sample MIDAS model file - the `sin` model.

3.2 A Sample Model

To illustrate the model file format, the model file for the model `sin` is shown in Figure 14. This model produces an output `y` which is the sine of the input `t`. Three parameters, `A`, `gain` and `f`, determine the amplitude and frequency. For more information on the `sin` model, see the model description at the end of this manual.

3.3 Pin Functions

As alluded to previously, there are several special purpose pin functions. These functions are used to obtain information about a pin and the signal connected to the pin. The syntax for these functions is

functionname(*pinname*)

where the *functionname* is the name of a pin function, and the *pinname* is the name of the model pin about which information is requested. The pin functions currently defined and the values they return are

`IsActive` – Returns true if the pin is connected to a signal.
`IsChangeable` – Returns true if the pin is connected to either a control signal or a net signal.
`IsConstant` – Returns true if the pin is connected to a constant.
`IsControl` – Returns true if the pin is connected to a control signal.
`IsNet` – Returns true if the pin is connected to a net signal.
`IsTriggered` – Returns true if the pin is connected to signal whose value has changed in the last simulation pass.
`Name` – Returns a pointer to a string containing the name of the signal to which the pin is connected.
`Stype` – Returns an integer that is equal to one of the constants `constanttp`, `controltp`, `nettp`, or `dummytp` depending on whether the signal connected to the pin is a constant, control signal, net signal, or nothing.

See Figure 15 for an example model which uses some of these pin functions.

3.4 Model Functions

Similar to pin functions, there are several special purpose model functions. The syntax for these functions is

functionname`()`

where the *functionname* is the name of a model function. The model functions currently defined are

`ContinueSimulation` – Tells MIDAS to make one more simulation pass. The simulation ends when no model calls this function. It is used mainly by models such as `time`.
`Enable` – Tells MIDAS to execute this model during the next simulation pass whether or not any inputs have changed. It is used mainly by models such as `time` to keep themselves going.
`IsEnabled` – Returns true if the model has had inputs which have changed. Useful only in the `LASTEXECUTE` section.
`IsFirstPass` – Returns true if this is the very first time the model has been initialized. Only valid in the `INITIALIZE` section.

```
(# A strange sample model - It initializes a to -1 in the
 # first simulation of the experiment, and 1 otherwise.
 # During execution, this model sets its output y to a if
 # the input x1 has changed or 2*a if only input x2 has
 # changed.
 #)

MODEL aSample

    double input x1, x2;
    double output y;
    double a;

INITIALIZE

    if (IsFirstPass()) a = -1; else a = 1;

EXECUTE

    if (IsTriggered(x1)) y = 1;
    else if (IsTriggered(x2)) y = 2;

END
```

Figure 15: MIDAS model file demonstrating pin and model functions.

For a sample model which uses some of these functions, see Figure 15.

3.5 Compiling the Model

Once the model file has been written, it must be compiled into MIDAS. This is done in two steps. First, the model file must be placed in MIDAS's `midas3/src/user` directory, and must have the suffix "`.m`". Next, from the `midas3/src` directory, execute the command `make models`. This recompiles MIDAS with the new model. If this is accomplished without error, the new model becomes available to netlists. See Appendix B for more details.

3.6 Performance Enhancement

MIDAS netlists are typically composed of many relatively primitive models, such as `add` and `delay`, that together constitute a complex systems. This approach provides a great deal of flexibility in a netlist to simulate a variety of different systems quickly and easily.

There are times, however, when it is desirable to runs very long simulations on a fixed architecture. For example, this may arise when evaluating the power of spectral tones as a dc signal is swept in fine increments across the input range of a $\Sigma\Delta$ modulator. For such experiments, it is advantageous to write a single model to describe a large portion of the system. With this approach, much of the overhead of calling numerous models is avoided and the simulation can be speeded up considerably. For an example of models that provide extended functionality, the reader is encouraged to examine the `sigma1` and `sigma2` models.

Furthermore, many compilers offer code optimization at the expense of a longer compilation time. In some systems, significant performance advantages can be obtained by using the compiler optimization functions. Consult your compiler manual for more details regarding this option.

3.7 Some Warnings

The model file compiler is actually implemented as a translator from a model file to a C++ file. That C++ file is then compiled into MIDAS. For this reason, most of the errors in model files are caught by the C++ compiler. Since the error handling capabilities of compilers vary, so will the error messages generated by erroneous model files. Some C++ compilers will not catch an error as until several lines after it occurred. When this occurs, the error will almost always be in the same section of the model file as the line caught by the compiler.

Occasionally, errors may occur that are not the fault of the model writer. Since the new model is being compiled into a larger program, sometimes errors like "function redefined" occur. When this happens, it means that the model accidentally contains a variable or function name used in the main program. The only solution is to rename the offending function or variable name in the model file.

Also, while assigning to input and parameter pins is said to be illegal in Section 3.1, there are circumstances in which this error will not be detected. The result is unpredictable.

One important point and a possible source of confusion is accessing vectors and matrices in models. When making an assignment to an element of a vector or matrix, the element must be accessed by enclosing the variable name in parentheses when

invoking the element. Thus, to access element 4 of vector `vec`, the following syntax must be used

```
(vec)[4] = x;
```

Note that writing to an element of a vector or matrix does not trigger the models connected to that output. This means that other models that use that pin will not necessarity be tagged to be executed in the next simulation pass. This degree of freedom is useful in some models. For example, in the `Nuttall30` windowing model, a vector is assigned to throughout a simulation. However, computations on this vector should occur only at the end of the simulation. To explicitly trigger models connected to the vector or matrix, it is necessary to make a dummy assignment of the variable to itself as follows

```
vec = vec;
```

One final word of caution should be mentioned regarding the assignment of integers to doubles and vice versa. The model writer must take care that variable assignments are carried out without type conflicts. This is not only true in MIDAS, but in all C programs. In previous versions of MIDAS, since no integer data types existed, it was unlikely that such problems would surface. In the new version of MIDAS, the model writer must be particularly aware of this possibility. As an illustration of this point, consider an integer variable, `int_var`, and a double variable, `double_var`. The following assignments yield unpredictable results

```
double_var = int_var; (# this is wrong!  #)

int_var = double_var; (# this is wrong!  #)
```

The correct form of the assignemt is to "typecast" the integer as a double when an assignment is made to `double_var` and to use the conversion utility function, `dtol`, when an assignement is made to `int_var`.

```
double_var = (double) (int_var); (# this is correct #)

int_var = dtol(double_var); (# this is correct #)
```

4 Writing Advanced Models

While the vast majority of models can be written using the techniques of Section 3, there are special cases which require that one bypass the model file translator and create the C++ file directly. These special cases include having pin arrays (similar to vectors in C++) and having pins whose type is determined at run-time by the signals to which the pins are connected. For a good example of these cases, the interested reader should study the code for the `print` model, which can be found in `midas3/src/mcore/EL` after reading this section.

In this section, we discuss the C++ file format. A working knowledge of C++ is assumed. For continuity, we use the C++ file that is generated by the model file `sin`, which is the example model file shown in Figure 14. The C++ file generated from this model file is reproduced in Figure 16.[3] The line numbers are supplied for reference only and are not actually part of the program. Note that in the C++ file, and everywhere else in the MIDAS source code, models are called "elements" for historic reasons.

The `sin` model has an output `y` which is the sine of the input `t`. Three parameters, `A`, `gain`, and `f`, determine the amplitude and frequency.

The purpose of lines 1 to 3 is to include the definitions of the underlying objects that define models and signals. They must be present in every model's C++ file. Line 4 includes the definitions needed for the `sin` function used in this model. The definition of the model `sin` is given in lines 6 to 16 in the form of an object declaration for `SinElement`. The implementation of this object is shown in lines 18 to 43.

The model pins are defined in lines 7 to 9. The variable types allowed in a pin definition are `InputSignal`, `ParameterSignal`, `OutputSignal`, and `LateOutputSignal`, corresponding to the pin types described in Section 3. They themselves are implemented as classes in C++ and declared in the header file `Elements.h`. Note that the pin variables are actually defined as pointers to signals. This will be important later. Line 10 defines internal model variables `a` and `omega`.

The operations on the object that defines the model `sin` are declared in lines 12 to 14. The code for the constructor, `SinElement::SinElement`, is shown in lines 18 to 26. Its purpose is to initialize all the variables declared in the object `SinElement`.

[3] This is not entirely true. Like many code generating computer programs, the model file translator generates some extraneous lines. The C++ file shown here is how the model file for `sin` should be translated.

```
 1  #include "def.h"
 2  #include "BaseElement.h"
 3  #include "Elements.h"
 4  #include <math.h>
 5
 6  class SinElement : public BaseElement {
 7      InputSignal *t;
 8      OutputSignal *y;
 9      ParameterSignal *A, *gain, *f;
10      double a, omega;
11  public:
12      SinElement(char*, ElementClass);
13      void Initialize();
14      void Execute();
15      BaseElement* Clone();
16  };
17
18  SinElement::SinElement(char* elname, ElementClass alg) :
19              BaseElement(elname, alg) {
20      _AddElement(this, elname);
21      t = AddInput(this, "t", doubletp, 0.0);
22      y = AddOutput(this, "y", doubletp);
23      A = AddParameter(this, "A", doubletp, 1.0);
24      gain = AddParameter(this, "gain", doubletp, 0.0);
25      f = AddParameter(this, "f", doubletp, 1.0);
26  }
27
28  void SinElement::Initialize() {
29      a = (A->Dbl()) * pow(10.0, (gain->Dbl())/20.0);
30      omega = 2.0 * M_PI * (f->Dbl());
31  }
32
33  void SinElement::Execute() {
34      (*y) = a * sin(omega * (t->Dbl()));
35  }
36
37  void SinElement::LastExecute() {
38      if (IsEnabled()) Execute();
39  }
40
41  BaseElement* SinElement::Clone() {
42      return new SinElement(Name(), algebraic);
43  }
44
45  SinElement Sin("sin", algebraic);
```

Figure 16: The C++ file for the model `sin`.

The function `_AddElement` is used at the beginning of the program to tell MIDAS that the model exists. The arguments to `_AddElement` are always `this` and `elname`. The other `Add...` functions are used when the net list is being processed to connect the model pins to the net. The possible `Add...` functions are `AddInput`, `AddParameter`, `AddOutput`, and `AddLateOutput`. There must be one such function for each pin. The first argument to the `Add...` function is always `this`. The second argument is a string which defines the pin name. By convention, this name is the same as the pin's variable name in the C++ file. All pin names for a particular model must be unique, but the pin names of different models may be the same. The third (optional) argument is the signal value type. Note that names for the signal value types used here are slightly different from those used in the model files, so that conflicts with C++ are avoided. The possible names are: `doubletp`, `integertp`, `complextp`, `vectp`, `mattp`, `strtp`, and `filetp`. The default is `doubletp`. For input and parameter signals of type `doubletp`, a default value can optionally be specified as a fourth argument.

The function `Initialize` is run once in the first pass of each simulation. In this case, it initializes the values of `a` and `omega`.

The function `Execute` determines the actual function of the model, which in this case is the computation of the sine of `t`. The value of a pin is returned by one of the following functions: `Dbl()`, `Int()`, `Cpl()`, `Vec()`, `Mat()`, `Str()`, and `File()`, depending on its type. Since the inputs and parameters for this model are pins of type `doubletp`, the appropriate function is `Dbl()`. The construct `t->Dbl()`, for example, returns the current value of signal connected to the pin `t`. The `->` operator is used because `t` is a pointer. The value of an output or late-output pin is set by a simple assignment to that pin. The C++ "overloading" mechanism is invoked on this assignment to let MIDAS know that this particular signal has changed. The MIDAS scheduler will then execute all models with inputs connected to this signal during the current or the next sampling period, depending on whether the pin is an output or a late-output.

While it is not used in the `sin` example, the function `LastExecute` may also be defined. It is executed at the end of each simulation. The `Initialize` and `LastExecute` functions are optional; the `Execute` function is not.

The `Clone` function (lines 15 and 41 to 43) is used by MIDAS to create instantiations of the model for each occurrence in the input deck. Line 45 makes the model known to the simulator and gives it a name. Each model must be given a unique name; a run-time error occurs otherwise. The "`algebraic`" which appears in lines 42 and 45 defines the model to be algebraic. If the model is nonalgebraic, these instances of `algebraic` should be replaced by `nonalgebraic`. Models which must *not* be algebraic are those which have a late output pin, those which trigger themselves with

the `Enable` function, and those which contain a `LastExecute` section. However, no check is made to insure that these rules are followed.

References

[1] R. Gregorian and G. Temes, *Analog MOS Integrated Circuits for Signal Processing*, Wiley, New York, 1986.

[2] J. Candy and G. Temes, *Oversampling Methods for A/D and D/A Conversion*, IEEE Press, New York, 1992.

[3] J. Candy, "A use of double integration in sigma delta modulation," *IEEE Trans. Commun.*, vol. COM-33, pp. 249–258, March 1985.

[4] B. Boser and B. Wooley, "The design of sigma-delta modulation analog-to-digital converters," *IEEE J. Solid-State Circuits*, vol. SC-23, pp. 1298–1308, December 1988.

[5] D. Su and B. Wooley, "A CMOS oversampling D/A converter with a current-mode semidigital reconstruction filter," *IEEE J. Solid-State Circuits*, vol. SC-28, pp. 1224–1233, December 1993.

[6] J. Candy B. Wooley and O. Benjamin, "A voiceband codec with digital filtering," *IEEE Trans. Commun.*, vol. COM-29, pp. 815–830, June 1981.

[7] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Redding and Menlo Park, 1987.

[8] *UNIX System V Release 2.0. User Reference Manual*, AT&T Bell Laboratories, Murray Hill, New Jersey, 1983.

Appendices

A What's New in MIDAS 3.1

This section outlines the differences between MIDAS 3.1 and MIDAS 2.1. The most important enhancement in the new version of MIDAS is the addition of an integer data type. Integers are useful in simulating digital systems. For example, the effects of truncation and overflow can be readily evaluated in a digital noise-shaper when using integer net signals in conjunction with models that operate on them. There are a few limitations when using integers. The MIDAS integer data type is defined as a `long integer`. On most machines, a `long integer` is limited to 32 bits. Moreover, only net signals may be integers. In other words, integers may be generated only by models in the `NETLIST` section. They may not be generated in the `CONSTS` or `CONTROL` sections. In order to be able to use constants and control signals as parameter inputs to integers models, all parameters of integer models supplied with the MIDAS distribution are defined as doubles. They are converted to integers within the models.

The existance of integers in MIDAS 3.1 requires an added level of diligence on the part of the model writer. As discussed in Section 3.7, assignment of integers to doubles and vice versa must be done using "typecasting" or utility functions in order to avoid type conflicts. See any book on C or C++ programming for more details on resolving type conflicts.

In the past, differing complex number libraries on various platforms were a source of portability problems. MIDAS 3.1 includes its own complex library to avoid these problems. Likewise, differences in the random number generation routines in various systems have prompted a new random number header file that can easily be changed to reconfigure MIDAS for various platforms. See Appendix B for more information. Some system calls that proved to be portability problems, such as those found in `time.h`, have also been eliminated.

The file inclusion feature via the `include` command has been enhanced so that it can utilize the standard UNIX file expansion character, $\sim$. Furthermore, the output of MIDAS 3.1 is along `stdout` as well as `stderr` to make it easier to separate the netlist echo and error messages from the desired output. The MIDAS 3.1 distribution includes 86 new models and 20 new example input files. Finally, the MIDAS 3.1 manual includes several relatively elaborate examples to help users become familiar with the capabilities of the simulator quickly.

B Installation

The source code for MIDAS is available on tape and 3.5" diskettes in UNIX tar format [8], by Email (midas@par.Stanford.EDU), and via the following World Wide Web site

```
http://cis.stanford.edu/icl/wooley-grp/midas.html
```

In addition to standard UNIX utilities, a C++ compiler[4] is needed [7]. The installation process under UNIX is outlined below. First, mount the tape and go to the directory you want to place MIDAS in, then type

```
tar x
```

If untarring from a file, type

```
tar xf <filename>
```

A directory named `midas3` will be created along with several subdirectories that contain the source code, documentation, and example files. To create an executable file for the first time, go to the directory `midas3/src` and type

```
make install
```

The compilation will take some time. An executable file for MIDAS will be created and stored in the directory `midas3/bin`. If using a C++ compiler with a name other than `CC`, specify the compiler name on the make command line. For example, when compiling with the GNU C++ compiler, `g++`, use

```
make install CC=g++
```

To compile a new user defined model, place the model in `midas3/src/user` and, in `midas3/src`, type

```
make models
```

Be sure to specify the C++ compiler name if it differs from `CC`.

Note that many compilers have options for execution speed optimization. On some systems, setting a high level of optimization can result in significantly faster execution rates. Compiler flags can be passed to the `make` program using the `AFLAGS` variable on the command line. Consult your compiler manual regrading optimization options.

[4] C++ is available without charge from the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307 (telephone 617-542-5942, Email gnu@prep.ai.mit.edu). C++ is also available from other vendors including AT&T, Software Sales and Marketing, P.O. Box 25000, Greensboro, NC 27420 (telephone 800-828-UNIX).

The MIDAS distribution includes an `examples` directory, which has two subdirectories, `simple` and `advanced`. These directories contain the MIDAS input files shown in this manual as well as many other examples to help the new user come up to speed quickly. These examples are supplied purely as a courtesy to the user and in no way imply suitability for a particular application.

As discussed in Section 1.2, the output of MIDAS 3.1 is along two streams, `stdout` and `stderr`. This makes it easy to separate the netlist echo and error messages from the output data. Previous versions of MIDAS had their outputs only along the `stdout` stream. For backward compatibility, there is a simple provision to change the MIDAS output streams back to only `stdout`. Edit the file `err.h` in the directory `midas3/src/mcore/include` and change the definition of `ErrFile` from `stderr` to `stdout`. Then, reinstall the program by executing `make` as discussed above.

On some systems, the random number generation library functions, `random` and `srandom` are not available. Instead, the `rand` and `srand` functions may exists. MIDAS models that use random numbers, by default, employ the former set of functions due to their superior statistical properties. The system can easily be reconfigured, however, to use the latter set of function calls. To do this, edit the file `randnums.h` in the directory `midas3/src/mcore/include` and change the definition of `Rand_Call` and `Rand_Seed` from `random` and `srandom` to `rand` and `srand`, respectively. Then, reinstall the program by executing `make` as discussed above.

The documentation for MIDAS is kept in the directory `midas3/manual`. A postscript file is available for immediate printing to a postscript printer. The manual can also be generated from the source code, which is located in `midas3/manual/doc`. All documents are formated for the typesetting language LaTeX. To generate this manual, switch to the directory `midas3/doc/man` and execute the `make` command. The output file, `root.dvi`, is generated in TeX's `DVI` format. Consult your local TeX wizard for information regarding printing this file.

C Error Messages

MIDAS has extensive error detection and reporting capabilities for both input parsing and simulation. The error messages and their meaning are listed below. Note that, for historic reasons, the error messages use the term "element" instead of "model". The two terms are equivalent.

`Cannot open file` *filename*
: The operating system was unable to open the file *filename*. Usually occurs when an attempt was made to read a non-existent file.

`Command line parameter #`*paramno* `not passed`
: The *paramno*'th command line parameter was not specified when MIDAS was executed, and the expression $*paramno* in the input file caused MIDAS to expect that parameter.

`Constant` *constantname* `already defined`
: Constants may only be assigned to once. An attempt was made to reassign the constant *constantname*.

`Constant` *constantname* `driven by output` *pinname*
: An attempt was made to connect the output pin *pinname* to the constant *constantname*. Output pins may only be connected to net signals.

`Constant` *constantname* `is not defined`
: The constant *constantname* has not been assigned a value in the input file lines above the one in which this error occurred.

`Constant` *constantname*`:  illegal type (real expected)`
: Only real numbers are allowed in expressions. The constant *constantname* must have the signal value type `double`, i.e., have been assigned to a real number.

`Control signal` *controlname* `already defined`
: Control signals may only appear in one `FOR` statement. The control signal *controlname* appears in more than one `FOR` statement.

`Control signal` *controlname* `driven by output` *pinname*
: An attempt was made to connect the output pin *pinname* to the control signal *controlname*. Output pins may only be connected to net signals.

`Division by zero`
: One of the constants in the denominator of a constant expression division evaluated to zero.

`Element` *modelname* `is not defined`
: The model *modelname* is not a standard or user-defined model.

`Illegal logarithmic range`
: A control range using the `LOG` form has a start and/or stop value which is not a positive number.

`Netlist cycle involving the signals:` *signallist*
: There is a loop involving the signals in *signallist* which does not contain a delay. All feedback loops in MIDAS must contain a delay.

`Non-constant` *signalname* `used in an expression`
: Only constants can appear in arithmetic expressions. To perform arithmetic on control or net signals, the models `add`, `sub`, `mul`, and `div` must be used.

`Out of memory (heap) --- buy a bigger computer!`
: Bad news. Maybe it is possible to alter the parameters of the simulation so that less memory is needed.

`Pin` *pinname* `is an input pin, not an output pin`
: Use the connector `<-` for this pin, not `->`.

`Pin` *pinname* `is an output pin, not an input pin`
: Use the connector `->` for this pin, not `<-`.

`Pin` *pinname* `multiply defined`
: A model pin can only be connected to one signal. This error occurs when the pin *pinname* is connected more than once in the same list of pin connections.

`Pin` *pinname* `not defined for element` *modelname*
: The pin *pinname* is not part of the model *modelname*. See the model's documentation for the pins which are.

`Signal` *signalname* `in element` *modelname* `is not invariant`
: All parameters must be connected to invariant signals. The signal *signalname* in the element *modelname* is connected to a parameter and is not invariant.

`Signal` *signalname* `is not driven by an output`
: The signal *signalname* is not a constant, a control signal, or driven by the output of a model.

`Syntax error in file constant`
: See Section 2 for the proper file constant syntax.

`Two sources defined for signal` *signalname*
: An attempt was made to connect the net signal *signalname* to more than one output pin. A net signal may only be driven by one output pin.

`Type mismatch between signal` *signalname* `and pin` *pinname*
: The signal value type for the signal *signalname* must match that expected by the pin *pinname*. See the model's documentation for the value type expected by each pin.

The following errors occur only when there are new user defined models.

`Element` *modelname* `defined twice`
: An attempt was made to add a user defined model with the same name as a previously defined model.

`Function Clone() not defined for element` *modelname*
: The Clone() function is not defined in the C++ file for the model *modelname*.

`Function Execute() not defined for element` *modelname*
: The Execute() function is not defined in the C++ file for the model *modelname*.

Errors which are not listed in this appendix indicate a bug in MIDAS.

D Model Library

This appendix describes the standard models supplied with MIDAS. They include a variety of models that are useful for describing and analyzing linear and nonlinear systems, as well as models that are used to generate inputs and outputs. Additional models, for example to perform special analyses not in the standard version of MIDAS, can be added by the user as discussed in Sections 3 and 4.

The table below lists all of the standard models, classified by their use. Detailed model documentation then follows in alphabetical order sorted by the names of the models. Unless otherwise indicated, the signal type expected by the pins of a model is `double`. If a default value is not indicated for an input or a parameter, it is assumed to be zero. No `PARAMETER` pins are of the type `integer` even in models that are otherwise purely integer models. Parameters are converted to integers within some models. This was done so that the `CONST` and `CONTROL` sections, which do not support integers, can be used to set parameter pins of integer models. The nonalgebraic models are identified with a "†".

- **Arithmetic operations:**

  `add`, `add4`, `sub`, `mul`, `mul4`, `div`, `neg` – Floating point functions.
  `addint`, `add4int`, `subint`, `mulint`, `mul4int`, `divint`, `negint` – Integer functions.
  `addintN`, `add4intN`, `subintN`, `mulintN`, `mul4intN`, `divintN` – N-bit integer functions.

- **Functions:**

  `double2int`, `int2double`, `double2vec`, `matrix2vec` – Type conversion.
  `double2N`, `int2N` – Conversion to N-bit integers.
  `ifequal`, `ifgreater`, `ifelse`, `ifgtelse`, `mux2` – Conditional assignment.
  `abs`, `dB`, `exp`, `ln`, `pow`, `sqrt` – Mathematical functions.
  `absint`, `powint`, `shift` – Integer mathematical functions.
  `zins`, `zoh`, `zinsint`, `zohint` – Zero insertions and zero-order holds.
  `sigma1`†, `sigma2`† – Sigma-delta modulators.
  `distortDiff`, `distortIter`, `settle` – Miscellaneous functions.

- **Estimation:**

  `mean`†, `power`†, `variance`† – Statistical functions.

`min`†, `mininvec`†, `mininmat`†, `max`†, `maxinvec`†, `maxinmat`† – Extrema.
`minint`†, `maxint`† – Integer extrema.

- **Quantization:**

`limiter`†, `limiterint`† – Hard clippers.
`quant`, `quantint` – Two level quantizers.
`quantizer`, `quantizerint` – User defined quantizers.
`quantUniform`, `quantUniformint` – Uniform quantizers.
`quantnldac`, `quantnlave` – Quantizers with random mismatches.

- **Function generators:**

`zeros`, `ones`, `consts` – Constant output sources.
`impulse`, `impulseint` – Kronecker deltas.
`sin`, `cos`, `sinzins`, `sinzoh` – Sinusoidal sources.
`sinN`, `sinNzins`, `sinNzoh` – N-bit sinusoidal sources.
`pulse` – Pulse generator.
`divider` – Square wave generator.
`gauss`, `uniform` – Random number generators.
`time`†, `timeint`† – Time base for simulation.

- **Filters:**

`decimate`, `decimateint` – Sub-samplers.
`delay`†, `delayint`†, `pureDelay`† – Delay elements.
`clkDelay`†, `clkDelayint`† – Triggerable delay elements.
`iir`, `iirint` – Infinite impulse response filters.
`linfir`, `linfirint` – Linear phase filters.
`sinc3`, `sinc3int`, `sinc4`, `sinc4int` – Comb filters.
`sincN`, `sincNint` – Nth order comb filters.
`skip`, `skipint` – Skip initial transient of a signal.
`tapDelay`†, `tapDelayint`† – Tapped delay lines.

- **Analyses:**

`disto`†, `intermod`† – Distortion analysis.
`dft`†, `fft`† – Discrete Fourier transform.
`rectangular`† – Rectangular windowing function.
`Blackman`†, `BHarris`†, `Hamming`†, `Hanning`†, `Nuttall30`† – Windowing functions.

`pdf`† – Probability density.
`pad`† – Pad a sequence with a constant.

- **I/O Primitives:**

  `doubleIn`† – Read floating point numbers from file.
  `integerIn`† – Read integers from file.
  `complexIn`† – Read complex numbers from file.
  `vectorIn`† – Read a vector from file.
  `matrixIn`† – Read a matrix from file.
  `hexin`† – Hexadecimal input from file.
  `monitor` – Print name and value of signal.
  `print` – Formated output.

NAME

`abs` – Absolute value.

INPUT

`x`

OUTPUT

`y`

FUNCTION

$y = |x|$

EXAMPLE

The model file line:

```
abs (x<-(-4.7), y->out);
```

sets the value of `out` to 4.7.

NAME

`absint` – Absolute value of an integer.

INPUT

`x` – An integer.

OUTPUT

`y` – An integer.

FUNCTION

$y = |x|$

EXAMPLE

The model file line:

```
absint (x<-(-4), y->out);
```

sets the value of `out` to 4.

NAME

`add` – Output `y` is the sum of inputs `x1` and `x2`.

INPUTS

`x1`, `x2`

OUTPUT

`y`

FUNCTION

$y = x1 + x2$

EXAMPLE

```
add (x1<-a, x2<-b, y->c);
```

NAME

`add4` – Output `y` is a weighted sum of up to four inputs.

INPUTS

`x1`, `x2`, `x3`, `x4`

OUTPUT

`y`

PARAMETERS

`a1`, `a2`, `a3`, `a4` – Weighting factors. Default: 1.

FUNCTION

$y = a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_4$

EXAMPLE

The model line:

```
add4 (x1<-a, a2<-(-2), x2<-b, a3<-3, x3<-c, y->y);
```

performs the function:

$$y = a - 2b + 3c$$

NAME

`add4int` – Output `y` is the integer weighted sum of up to four integers.

INPUTS

`x1`, `x2`, `x3`, `x4` – Integers.

OUTPUT

`y` – An integer.

PARAMETERS

`a1`, `a2`, `a3`, `a4` – Weighting factors. Default: 1.

FUNCTION

$y = a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_4$

EXAMPLE

The model line:

```
add4int (x1<-a, a2<-(-2), x2<-b, a3<-3, x3<-c, y->y);
```

performs the function:

$$y = a - 2b + 3c$$

NAME

`add4intN` – Output `y` is the n-bit, 2's complement weighted sum of up to four integers.

INPUTS

`x1`, `x2`, `x3`, `x4` – Integers.

OUTPUT

`y` – The n-bit integer output.
`yfull` – The non-truncated integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.

PARAMETERS

`a1`, `a2`, `a3`, `a4` – Weighting factors. Default: 1.
`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.

FUNCTION

If the sum of the weighted inputs is greater than the maximum, $2^{n-1}-1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

BUG

The maximum value of `n` is machine dependent and is typically 31. If a greater value is specified, the model automatically reduces `n`, without warning, to the maximum supported by the platform on which the program is running.

NAME

`addint` – Output `y` is the integer sum of integer inputs `x1` and `x2`.

INPUTS

`x1`, `x2` – Integers.

OUTPUT

`y` – An integer.

FUNCTION

$y = x1 + x2$

EXAMPLE

```
addint (x1<-a, x2<-b, y->c);
```

NAME

`addintN` – Output `y` is the n-bit, 2's complement sum of integer inputs `x1` and `x2`.

INPUTS

`x1`, `x2` – Integers.

OUTPUT

`y` – The n-bit integer output.
`yfull` – The non-truncated integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.

PARAMETERS

`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.

FUNCTION

If the sum of the inputs is greater than the maximum, $2^{n-1}-1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

BUG

The maximum value of `n` is machine dependent and is typically 31. If a greater value is specified, the model automatically reduces `n`, without warning, to the maximum supported by the platform on which the program is running.

NAME

`BHarris` – A scaled Blackman-Harris window. Intended for use with the `fft` and `dft` models.

INPUT

`x` – Input samples, floating point numbers.
`xint` – Input samples, integers.

OUTPUTS

`w` – A vector containing the windowed and scaled input samples, produced at the end of the simulation.
`N` – The size of `w`, i.e., the number of samples used, an integer.

PARAMETERS

`refernce` – Reference level for power: amplitude of sinusoid corresponding to 0 dB. Default: 0.5.
`DFT` – A flag indicating whether the output will be used with the `fft` or the `dft` model. If `DFT` is 0 the length of the output sequence is truncated to a power of 2, otherwise it is unaltered. Default: 0.

FUNCTION

This model first stores a sequence of simulation results as a vector of input samples. The `x` input pin should be used if the input is a real number and the `xint` input pin should be used if the input in an integer. Upon completion of the simulation, depending on the value of the `DFT` parameter, this vector is either left unchanged or truncated so as to contain the first 2^n samples, where n is an integer between 1 and a machine dependent value, typically 31. Then these samples are windowed using the Blackman-Harris window. Finally, the windowed samples are scaled such that when the FFT or DFT is taken, the result will be the spectral power of the input normalized to the power of a sinewave of amplitude given by the `reference` parameter.

EXAMPLE

The model lines:

```
BHarris (x<-x, w->x_win);
fft (x<-x_win, fs<-fs, S->S);
```

```
print (x1<-S);
```

print an estimate of the spectrum of `x`, which has a sampling rate of `fs`.

REFERENCE

A. Oppenheim and R. Schafer, "Discrete-Time Signal Processing," Prentice Hall, 1989.

NAME

`Blackman` – A scaled Blackman window. Intended for use with the `fft` and `dft` models.

INPUT

`x` – Input samples, floating point numbers.
`xint` – Input samples, integers.

OUTPUTS

`w` – A vector containing the windowed and scaled input samples, produced at the end of the simulation.
`N` – The size of `w`, i.e., the number of samples used, an integer.

PARAMETERS

`refernce` – Reference level for power: amplitude of sinusoid corresponding to 0 dB. Default: 0.5.
`DFT` – A flag indicating whether the output will be used with the `fft` or the `dft` model. If `DFT` is 0 the length of the output sequence is truncated to a power of 2, otherwise it is unaltered. Default: 0.

FUNCTION

This model first stores a sequence of simulation results as a vector of input samples. The `x` input pin should be used if the input is a real number and the `xint` input pin should be used if the input in an integer. Upon completion of the simulation, depending on the value of the `DFT` parameter, this vector is either left unchanged or truncated so as to contain the first 2^n samples, where n is an integer between 1 and a machine dependent value, typically 31. Then these samples are windowed using the Blackman window. Finally, the windowed samples are scaled such that when the FFT or DFT is taken, the result will be the spectral power of the input normalized to the power of a sinewave of amplitude given by the `reference` parameter.

EXAMPLE

The model lines:

```
Blackman (x<-x, w->x_win);
fft (x<-x_win, fs<-fs, S->S);
```

```
print (x1<-S);
```

print an estimate of the spectrum of `x`, which has a sampling rate of `fs`.

REFERENCE

A. Oppenheim and R. Schafer, "Discrete-Time Signal Processing," Prentice Hall, 1989.

NAME

`clkDelay` – A delay element triggered by a clock input.

INPUTS

`x1`, `x2`
`clk` – Trigger.

OUTPUTS

`y` – Delayed (late) output.
`v` – Immediate output.

PARAMETERS

`x0` – Initial state ($y[0]$). Default: 0.
`c1`, `c2` – Coefficients. Default: 1.

FUNCTION

When `clk` is greater than 0.5:

$$y[k+1] = c1 \cdot x1[k] + c2 \cdot x2[k]$$
$$v[k] = c1 \cdot x1[k] + c2 \cdot x2[k]$$

otherwise, the outputs do not change.

NAME

`clkDelayint` – A delay element that is triggered by a clock input and whose inputs and outputs are integers.

INPUTS

`x1`, `x2` – Integers.
`clk` – Trigger, an integer.

OUTPUTS

`y` – Delayed (late) output, an integer.
`v` – Immediate output, an integer.

PARAMETERS

`x0` – Initial state ($y[0]$). Default: 0.
`c1`, `c2` – Coefficients. Default: 1.

FUNCTION

When `clk` is not equal to 0:

$$y[k+1] = c1 \cdot x1[k] + c2 \cdot x2[k]$$
$$v[k] = c1 \cdot x1[k] + c2 \cdot x2[k]$$

otherwise, the outputs do not change.

NAME

`complexIn` – A clock whose output is data read from an input file.

PARAMETER

`file` – The input file.

OUTPUT

`y` – A complex number.

FUNCTION

The input data is read in as pairs of double precision, floating point numbers, the first of which becomes the real part and the second the imaginary part of the complex number `y`. `y` then acts as a clock signal for the netlist. Therefore, there is no need for a time base model.

EXAMPLE

The input file:

```
NETLIST
  complexIn (file<- <data.dat>, y->num);
  print (x1<-num, x2<-"\n");
END
```

prints the contents of the file "`data.dat`" to the standard output as complex numbers.

NAME

`consts` – Output a constant whenever the input changes.

INPUTS

`t`

OUTPUTS

`y` – Output samples, floating point numbers.
`yint` – Output samples, integers.

PARAMETERS

`constant` – Output value. Default: 0.

FUNCTION

The outputs `y` and `yint` are set equal to `constant` and triggered whenever the input changes. If `constant` is a real number, it is rounded to the nearest integer when `yint` is set equal to it.

NAME

`cos` – Sampled sinusoidal waveform generator.

INPUT

`t` – Time.

OUTPUT

`y`

PARAMETERS

`A` – Amplitude of sinusoid with gain at 0dB. Default: 1.0.
`gain` – Modify amplitude of sinusoid by `gain` dB. Default: 0.
`f` – Frequency. Default: 1.0.
`phase` – Phase in radians. Default: 0.0.

FUNCTION

$y = A \cdot 10^{gain/20} \cdot \cos(2\pi ft + phase)$

EXAMPLE

The model lines:

```
cos (t<-1/2000, A<-5, f<-1000, y->Out);
monitor (x<-Out);
```

print "`Out = -5.000`".

NAME

`dB` – Convert the input to dB.

INPUT

`x`

OUTPUT

`y`

PARAMETER

`a` – Conversion factor. Default: 10.

FUNCTION

$y = a \log_{10}(x)$

EXAMPLE

The model line:

```
db (x<-10e5, y->y);
```

sets the output `y` to 50.

BUG

The input `x` is clipped to a minimum value of 10^{-30}.

NAME

`decimate` – Decimator.

INPUT

`x`

OUTPUT

`y`

PARAMETER

`M` – Decimation ratio. Default: 1.

FUNCTION

Signal `y` is `x` sampled at 1/`M` times the rate of `x`. The first sample of `x` is always output, then `M`-1 samples are skipped before the next is output.

NAME

`decimateint` – Decimator for integer signals.

INPUT

`x` – An integer.

OUTPUT

`y` – An integer.

PARAMETER

`M` – Decimation ratio. Default: 1.

FUNCTION

Signal `y` is `x` sampled at 1/`M` times the rate of `x`. The first sample of `x` is always output, then `M`-1 samples are skipped before the next is output.

NAME

`delay` – Output `y` is the sum of inputs `x1` and `x2` delayed by one clock cycle.

INPUTS

`x1`, `x2`

OUTPUTS

`y` – Delayed (late) output.
`v` – Immediate output.

PARAMETERS

`x0` – Initial state ($y[0]$). Default: 0.
`c1`, `c2` – Coefficients. Default: 1.

FUNCTION

$y[k+1] = c1 \cdot x1[k] + c2 \cdot x2[k]$
$v[k] = c1 \cdot x1[k] + c2 \cdot x2[k]$

EXAMPLE

The model line:

```
delay (x1<-in, x2<-out, y->out, c1<-2, c2<-0.9)
```

creates a leaky integrator with the transfer function:

$$\frac{2 \cdot z^{-1}}{1 - 0.9 \cdot z^{-1}}.$$

NAME

`delayint` – Integer output `y` is equal to integer input `x` delayed by one clock cycle.

INPUT

`x` – An integer.

OUTPUTS

`y` – Delayed (late) output, an integer.
`v` – Immediate output, an integer.

PARAMETER

`x0` – Initial state ($y[0]$). Default: 0.

FUNCTION

$y[k+1] = x1[k]$
$v[k] = x1[k]$

EXAMPLE

```
delayint (x1<-input, y->output);
```

NAME

`dft` – Compute the discrete Fourier transform of a real sequence using the conventional DFT algorithm, not the FFT algorithm, so that the length of the input need not be a power of 2.

INPUT

`x` – A vector containing the input samples. Its size need not be a power of 2.

OUTPUTS

`A` – A $2 \times N$ matrix representing a plot of the magnitude of the DFT of the input sequence. The first column of the matrix contains the frequency; the second column contains the magnitude of the spectrum. N is half the size of the input vector, i.e., only half of the spectrum is output. However, since the magnitude of the DFT of a real sequence is symmetric about $N/2$, the other half of the spectrum is just a mirror image of this half.
`S` – A Matrix similar to `A`, but with the magnitude in units of dB.
`P` – A Matrix similar to `A` containing the phase of the input sequence.

PARAMETER

`fs` – The frequency at which the input `x` was sampled.

FUNCTION

This model computes the DFT of the input sequence. When used in conjunction with a window model, the result is an estimate of the spectrum of a sequence. This model executes much more slowly than the `fft` model and should be used only when truncating the length of the input to the nearest power of 2 results in an unacceptable loss of information.

EXAMPLE

The model lines:

```
Nuttall30 (x<-x, w->x_win, DFT<-1);
dft (x<-x_win, fs<-fs, S->S);
print (x1<-S);
```

print an estimate of the spectrum of `x`, which has a sampling rate of `fs`.

NAME

`disto` – Distortion analysis for noisy sinusoidal signals.

INPUT

`x` – Input samples.

OUTPUTS

`amplitude` – Vector with amplitude of fundamental and harmonics.

`power` – Vector with power of signal and harmonics in dB below reference.

`mean` – Mean value of signal `x` (excluded from `snr` and `mse` computations).

`snr` – Signal-to-noise ratio in dB, excluding harmonics.

`tsnr` – Signal-to-noise ratio in dB, including harmonics.

`hsnr` – Signal-to-noise ratio in dB, only harmonics.

`mse` – Mean squared error in dB below reference, excluding harmonics.

`tmse` – Mean squared error in dB below reference, including harmonics.

`hmse` – Mean squared error in dB below reference, only harmonics.

`thd` – Total harmonic distortion in percent ($100 \cdot \sigma_{hh}^2/\sigma_{xx}^2$).

`festimate` – Estimated signal frequency relative to sampling rate (when `fxT` was not specified).

PARAMETERS

`fxT` – Frequency of fundamental. Default: estimated from data.

`fs` – Sampling frequency. Default: 1.

`harmonics` – Number of harmonics to be computed. Default: 0, i.e. only the fundamental. Maximum: 60.

`tol` – Absolute tolerance of frequency estimate when `fxT` is not specified. Default: 10^{-6}.

`reference` – Reference level for power: amplitude of sinusoid corresponding to 0 dB. Default: 0.5.

BUGS

Numerically unstable if the input sequence is longer than about ten million samples.

REFERENCE

B. Boser, K. Karmen, H. Martin, and B. Wooley, "Simulating and Testing Oversampled Analog-to-Digital Converters," *IEEE Trans. on Computer-Aided Design*, vol. 7, June 1988.

NAME

`distortDiff` – Output `y` is a weighted sum of the difference of squared and cubed inputs.

INPUTS

`x1`, `x2`

OUTPUT

`y`

PARAMETERS

`alpha1` – Quadratic weighting factor. Default: 100×10^{-6}.
`alpha2` – Cubic weighting factor. Default: 10×10^{-6}.

FUNCTION

$y = alpha_1 \cdot (x_1^2 - x_2^2) + alpha_2 \cdot (x_1^3 - x_2^3)$

NOTES

This function is useful in simulating certain circuit nonlinearities such as capacitor voltage coefficients.

NAME

`distortIter` – Output `y` is an iterative solution to a cubic equation.

INPUTS

`xin`, `xfb`

OUTPUT

`y`

PARAMETERS

`alpha1` – Quadratic weighting factor. Default: 100×10^{-6}.
`alpha2` – Cubic weighting factor. Default: 10×10^{-6}.
`maxiter` – Maximum number of iterations. Default: 5.
`tol` – Convergence tolerance. Default: 10^{-6}.

FUNCTION

Solve for y where $y = xin + xfb - alpha_1 \cdot (y^2 - xfb^2) + alpha_2 \cdot (y^3 - xfb^3)$

NOTES

Iteration is performed either `maxiter` times or until the estimates converge to `tol`. This model is useful in simulating certain circuit nonlinearities, such as capacitor voltage coefficients, that appear in a feedback loop.

BUGS

The iterative solution may not converge if the weighting factors become too large. Other solutions to the above equation, such as Cardano's method, may be implemented.

NAME

`div` – Output `y` is the quotient of inputs `x1` and `x2`.

INPUTS

`x1`, `x2`

OUTPUT

`y`

FUNCTION

$y = x1/x2$

EXAMPLE

```
div (x1<-a, x2<-b, y->c);
```

BUGS

Proper error message is generated for division by zero, but not for overflow.

NAME

`divider` – Frequency divider.

INPUT

`x` – Reference frequency.

OUTPUT

`y`

PARAMETERS

`ym` – Negative output level. Default: -0.5.
`yp` – Positive output level. Default: 0.5.
`ratio` – Ratio of input (`x`) frequency to output (`y`) frequency. Default: 2.

FUNCTION

Square wave with levels `ym` and `yp` at a frequency which is 1/`ratio` times that of the input.

BUG

Accepts only integer ratios larger than or equal to 2. Any lower ratio specified will be rounded up to this value without warning. For odd ratios, the interval when `y` = `yp` is slightly longer than the one when `y` = `ym`.

NAME

`divint` – Integer output `y` is the quotient of integer inputs `x1` and `x2`.

INPUTS

`x1`, `x2` – Integers.

OUTPUT

`y` – An integer.
`remainder` – An integer.

FUNCTION

$y = x1/x2$

EXAMPLE

```
divint (x1<-a, x2<-b, y->c, remainder->d);
```

NAME

`divintN` – Integer output `y` is the quotient of integer inputs `x1` and `x2`.

INPUTS

`x1`, `x2` – Integers.

OUTPUT

`y` – An integer.
`yfull` – The non-truncated integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.
`remainder` – An integer.

PARAMETERS

`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.

FUNCTION

If the result of the division is greater than the maximum, $2^{n-1}-1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

BUG

The maximum value of `n` is machine dependent and is typically 31. If a greater value is specified, the model automatically reduces `n`, without warning, to the maximum supported by the platform on which the program is running.

NAME

`double2int` – Convert a double precision floating point number to an integer.

INPUT

`x` – A floating point number.

OUTPUT

`y` – An integer.

FUNCTION

Output `y` is equal to the input `x` rounded off to the nearest integer.

NAME

`double2N` – Output `y` is the n-bit, 2's complement conversion of floating point input `x`.

INPUTS

`x` – A floating point number.

OUTPUT

`y` – The n-bit integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.

PARAMETERS

`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.

FUNCTION

If the input is greater than the maximum, $2^{n-1}-1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

BUG

The maximum value of `n` is machine dependent and is typically 31. If a greater value is specified, the model automatically reduces `n`, without warning, to the maximum supported by the platform on which the program is running.

NAME

`double2vec` – Convert a sequence of floating point numbers to a vector.

INPUT

`x` – A floating point number.

OUTPUT

`y` – A vector containing the inputs, presented at the end of the simulation.

FUNCTION

Output `y` is a vector whose elements are the inputs `x`.

NAME

`doubleIn` – A clock whose output is data read from an input file.

PARAMETER

`file` – The input file.

OUTPUT

`y` – A double precision number.

FUNCTION

The output pin `y` is set equal to data read in from `file` as double precision, floating point numbers. `y` then acts as a clock signal for the netlist. Therefore, there is no need for a time base model.

EXAMPLE

The input file:

```
NETLIST
  doubleIn (file<- <data.dat>, y->num);
  print (x1<-num, x2<-"\n");
END
```

prints the contents of the file "`data.dat`" to the standard output as double precision, floating point numbers.

NAME

`exp` – Output `y` is an exponential with base e of input `x`.

INPUT

`x`

OUTPUT

`y`

FUNCTION

$y = e^x$

EXAMPLE

```
exp (x<-a, y->b);
```

NAME

`fft` – Compute the discreye Fourier transform of a real sequence using the FFT algorithm.

INPUT

`x` – A vector containing the input samples. Its size must be 2^n, where n is an integer greater than 1.

OUTPUTS

`A` – A $2 \times N$ matrix representing a plot of the magnitude of the DFT of the input sequence. The first column of the matrix contains the frequency; the second column contains the magnitude of the spectrum. N is half the size of the input vector, i.e., only half of the spectrum is output. However, since the magnitude of the DFT of a real sequence is symmetric about $N/2$, the other half of the spectrum is just a mirror image of this half.

`S` – A Matrix similar to `A`, but with the magnitude in units of dB.

`P` – A Matrix similar to `A` containing the phase of the input sequence.

PARAMETER

`fs` – The frequency at which the input `x` was sampled.

FUNCTION

This model computes the DFT of the input sequence. When used in conjunction with a window model, the result is an estimate of the spectrum of a sequence.

EXAMPLE

The model lines:

```
Nuttall30 (x<-x, w->x_win);
fft (x<-x_win, fs<-fs, S->S);
print (x1<-S);
```

print an estimate of the spectrum of `x`, which has a sampling rate of `fs`.

NAME

`gauss` – Gaussian random number generator.

INPUT

`t` – Clock; a random number is generated every time `t` changes.

OUTPUT

`y`

PARAMETERS

`mean` – Mean of the random numbers. Default: 0.
`stddev` – Standard deviation of random numbers. Default: 1.
`seed` – Seed for random number generator. Default: 1.

FUNCTION

Generates independent Gaussian random numbers.

EXAMPLE

The model lines:

```
time (k->kT, kstart<-1, kstop<-10);
gauss (t<-kT, y->y);
monitor (x<-y);
```

print 10 random numbers that have a gaussian distribution with a mean of 0 and a standard deviation of 1.

BUG

This model is based on the standard UNIX function "`random.c`"; results might be incorrect on other systems. See the UNIX manual for the properties of "`random`".

NAME

`Hamming` – A scaled Hamming window. Intended for use with the `fft` and `dft` models.

INPUT

`x` – Input samples, floating point numbers.
`xint` – Input samples, integers.

OUTPUTS

`w` – A vector containing the windowed and scaled input samples, produced at the end of the simulation.
`N` – The size of `w`, i.e., the number of samples used, an integer.

PARAMETERS

`refernce` – Reference level for power: amplitude of sinusoid corresponding to 0 dB. Default: 0.5.
`DFT` – A flag indicating whether the output will be used with the `fft` or the `dft` model. If `DFT` is 0 the length of the output sequence is truncated to a power of 2, otherwise it is unaltered. Default: 0.

FUNCTION

This model first stores a sequence of simulation results as a vector of input samples. The `x` input pin should be used if the input is a real number and the `xint` input pin should be used if the input in an integer. Upon completion of the simulation, depending on the value of the `DFT` parameter, this vector is either left unchanged or truncated so as to contain the first 2^n samples, where n is an integer between 1 and a machine dependent value, typically 31. Then these samples are windowed using the Hamming window. Finally, the windowed samples are scaled such that when the FFT or DFT is taken, the result will be the spectral power of the input normalized to the power of a sinewave of amplitude given by the `reference` parameter.

EXAMPLE

The model lines:

```
Hamming (x<-x, w->x_win);
fft (x<-x_win, fs<-fs, S->S);
```

```
print (x1<-S);
```

print an estimate of the spectrum of `x`, which has a sampling rate of `fs`.

REFERENCE

A. Oppenheim and R. Schafer, "Discrete-Time Signal Processing," Prentice Hall, 1989.

NAME

`Hanning` – A scaled Hanning window. Intended for use with the `fft` and `dft` models.

INPUT

`x` – Input samples, floating point numbers.
`xint` – Input samples, integers.

OUTPUTS

`w` – A vector containing the windowed and scaled input samples, produced at the end of the simulation.
`N` – The size of `w`, i.e., the number of samples used, an integer.

PARAMETERS

`refernce` – Reference level for power: amplitude of sinusoid corresponding to 0 dB. Default: 0.5.
`DFT` – A flag indicating whether the output will be used with the `fft` or the `dft` model. If `DFT` is 0 the length of the output sequence is truncated to a power of 2, otherwise it is unaltered. Default: 0.

FUNCTION

This model first stores a sequence of simulation results as a vector of input samples. The `x` input pin should be used if the input is a real number and the `xint` input pin should be used if the input in an integer. Upon completion of the simulation, depending on the value of the `DFT` parameter, this vector is either left unchanged or truncated so as to contain the first 2^n samples, where n is an integer between 1 and a machine dependent value, typically 31. Then these samples are windowed using the Hanning window. Finally, the windowed samples are scaled such that when the FFT or DFT is taken, the result will be the spectral power of the input normalized to the power of a sinewave of amplitude given by the `reference` parameter.

EXAMPLE

The model lines:

```
Hanning (x<-x, w->x_win);
fft (x<-x_win, fs<-fs, S->S);
```

```
print (x1<-S);
```

print an estimate of the spectrum of `x`, which has a sampling rate of `fs`.

REFERENCE

A. Oppenheim and R. Schafer, "Discrete-Time Signal Processing," Prentice Hall, 1989.

NAME

`hexin` – A clock whose output depends on the data read from an input file. Useful for reading test data for analysis by MIDAS.

PARAMETER

`file` – The input file. Data is encoded in hexadecimal format. Each line starts with the characters "`0x`", followed by 8 hexadecimal digits (0–9, A–F, a–f).

OUTPUT

`y` – Is -0.5 for input bit 0, $+0.5$ for input bit 1.

FUNCTION

The input data is taken as a stream of bits which define a square wave. The least significant bit of a data word represents the first output bit from the word, the most significant bit represents the last output bit from the word. The resulting stream of bits is sent to the output `y` as a clock with a 0 bit producing a value of -0.5 and a 1 bit producing a value of $+0.5$.

NAME

`ifelse` – Conditional execution. Output `y` is set equal to the input `x1` if and only if `a1` is equal to `a2`. Otherwise, `y` is set equal to the input `x2`.

INPUT

`x1`, `x2`
`a1`, `a2` – Default: 0.

OUTPUT

`y`

FUNCTION

If $a_1 = a_2$, then $y = x_1$, else $y = x_2$.

NAME

`ifequal` – Conditional execution. Output `y` is set to the input `x` if and only if `a1` is equal to `a2`.

INPUT

`x`
`a1`, `a2` – Default: 0.

OUTPUT

`y`

FUNCTION

If $a_1 = a_2$, then $y = x$.

EXAMPLE

The input file:

```
CONTROL
  FOR m = [0, 1, 2, 3];
NETLIST
  ifequal (a1<-m, a2<-2, x<-m, y->trig);
  print (trigger<-trig, x1<-"m is", x2<-m, x3<-"\n");
END
```

produces the output:

```
m is 2
```

NAME

`ifgreater` – Conditional execution. Output `y` is set equal to the input `x1` if and only if `a1` is greater than `a2`.

INPUT

`x`
`a1`, `a2` – Default: 0.

OUTPUT

`y`

FUNCTION

If $a_1 > a_2$, then $y = x_1$.

NAME

`ifgtelse` – Conditional execution. Output `y` is set equal to the input `x1` if and only if `a1` is greater than `a2`. Otherwise, `y` is set equal to the input `x2`.

INPUT

`x1`, `x2`
`a1`, `a2` – Default: 0.

OUTPUT

`y`

FUNCTION

If $a_1 > a_2$, then $y = x_1$, else $y = x_2$.

NAME

`iir` – An infinite impulse response filter.

INPUT

`x`

OUTPUT

`y`

PARAMETER

`a` – A vector containing the coefficients of the denominator in the format shown below. Note that the first coefficient is always set to 1 and the sign of the coefficients in vector `a` is reversed.

`b` – A vector containing the coefficients of the numerator.

`M` – Decimation ratio. Default: 1.

`skip` – Number of outputs to be suppressed from initial transient. Default: Filter order plus 1.

FUNCTION

If the length of vector `a` is M and the length of vector `b` is N, the filter transfer function is:

$$H(z) = \frac{\sum_{i=0}^{N-1} b_i \cdot z^{-i}}{1 - \sum_{i=0}^{M-1} a_i \cdot z^{-i}}$$

NAME

`iirint` – An infinite impulse response filter with integer input.

INPUT

`x` – An integer.

OUTPUT

`y`

PARAMETER

`a` – A vector containing the coefficients of the denominator in the format shown below. Note that the first coefficient is always set to 1 and the sign of the coefficients in vector `a` is reversed.

`b` – A vector containing the coefficients of the numerator.

`M` – Decimation ratio. Default: 1.

`skip` – Number of outputs to be suppressed from initial transient. Default: Filter order plus 1.

FUNCTION

If the length of vector `a` is M and the length of vector `b` is N, the filter transfer function is:

$$H(z) = \frac{\sum_{i=0}^{N-1} b_i \cdot z^{-i}}{1 - \sum_{i=0}^{M-1} a_i \cdot z^{-i}}$$

NAME

`impulse` – Kronecker delta at $t = t0$.

INPUT

`t` – Time.

OUTPUT

`y` – 1.0 at $t = t0$, 0.0 otherwise.

PARAMETER

`t0` – Time at which the impulse should be produced. Default: 0.

EXAMPLE

The model lines:

```
time (k->kT, kstop<-5);
impulse (t<-kT, t0<-2, y->imp);
monitor (x<-imp);
```

produce the sequence $\{0, 0, 1, 0, 0, 0\}$ at the output `imp`.

NAME

`impulseint` – Kronecker delta at $t = t0$.

INPUT

`t` – Time.

OUTPUT

`y` – Integer output: 1 at $t = t0$, 0 otherwise.

PARAMETER

`t0` – Time at which the impulse should be produced. Default: 0.

EXAMPLE

The model lines:

```
time (k->kT, kstop<-5);
impulseint (t<-kT, t0<-2, y->imp);
monitor (x<-imp);
```

produce the sequence $\{0,0,1,0,0,0\}$ at the output `imp`.

NAME

`int2double` – Convert an integer to a double precision floating point number.

INPUT

`x` – An integer.

OUTPUT

`y` – A floating point number.

FUNCTION

Output `y` is the double precision, floating point representation of the input `x`.

NAME

`int2N` – Output `y` is the n-bit, 2's complement conversion of integer input `x`.

INPUTS

`x` – An integer.

OUTPUT

`y` – The n-bit integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.

PARAMETERS

`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.

FUNCTION

If the input is greater than the maximum, $2^{n-1} - 1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

BUG

The maximum value of `n` is machine dependent and is typically 31. If a greater value is specified, the model automatically reduces `n`, without warning, to the maximum supported by the platform on which the program is running.

NAME

`integerIn` – A clock whose output is data read from an input file.

PARAMETER

`file` – The input file.

OUTPUT

`y` – An integer.

FUNCTION

The output pin `y` is set equal to data read in from `file` as integers. `y` then acts as a clock signal for the netlist. Therefore, there is no need for a time base model.

EXAMPLE

The input file:

```
NETLIST
  integerIn (file<- <data.dat>, y->num);
  print (x1<-num, x2<-"\n");
END
```

prints the contents of the file "`data.dat`" to the standard output as integers.

NAME

`intermod` – Intermodulation distortion analysis for noisy sinusoidal signals.

INPUT

`x` – Input samples.

OUTPUTS

`amplitude` – Vector with amplitude of test signals and intermodulation products.

`power` – Vector with power of test signals and intermodulation products in dB below reference.

`freq` – Vector with frequencies corresponding to the outputs of `amplitude` and `power`.

`mean` – Mean value of signal `x` (excluded from `snr` and `mse` computations).

`snr` – Sum of signals-to-noise ratio in dB, excluding intermodulation products.

`tsnr` – Sum of signals-to-noise ratio in dB, including intermodulation products.

`hsnr` – Sum of signals-to-noise ratio in dB, only intermodulation products.

`mse` – Mean squared error in dB below reference, excluding intermodulation products.

`tmse` – Mean squared error in dB below reference, including intermodulation products.

`hmse` – Mean squared error in dB below reference, only intermodulation products.

`thd` – Total harmonic distortion in percent ($100 \cdot \sigma_{hh}^2/\sigma_{xx}^2$).

PARAMETERS

`f1` – Frequency of a test signal. Default: estimated from data.

`f2` – Frequency of another test signal. Default: estimated from data.

`fs` – Sampling frequency. Default: 1.

`IMorder` – Order of intermodulation products to be computed. Default: 0, i.e. only the test signals. Maximum: 60.

`bandl` – Lower frequency of interest for searching for intermodulation products. Default: 0.

`bandh` – Upper frequency of interest for searching for intermodulation products.

`tol` – Tolerance of frequency estimate when `f1` or `f2` are not specified. Default: 10^{-6}.

`reference` – Reference level for power: amplitude of sinusoid corresponding to 0 dB. Default: 0.5.

FUNCTION

This model is used to find the intermodulation products of two sinusoids when passed through a nonlinear system. `bandl` and `bandh` specify a frequency window in which to evaluate intermodulation products and `IMorder` specifies the order of interest of intermodulation. This model is based on a modified version of the algorithm used in the `disto` model.

BUGS

Numerically unstable if the input sequence is longer than about ten million samples.

NAME

`limiter` – Hard clipper.

INPUT

`x`

OUTPUT

`y`
`yd` – Late (delayed) output.

PARAMETERS

`x0` – Initial state ($yd[0]$). Default: 0.
`min`, `max` – Signal limits. Default: $\pm\infty$.

FUNCTION

if $x >$ max then $y =$ max else
if $x <$ min then $y =$ min else
$yd = y = x$

EXAMPLE

The model line:

```
limiter (x<-input, y->output, min<-0)
```

limits the signal `output` to positive values of the signal `input`.

NAME

`limiterint` – Hard clipper.

INPUT

`x` – An integer.

OUTPUT

`v` – An integer.
`y` – Late (delayed) output, an integer.

PARAMETERS

`x0` – Initial state ($y[0]$). Default: 0.
`min`, `max` – Signal limits. Default: $\pm\infty$.

FUNCTION

if $x >$ max then $y =$ max else
if $x <$ min then $y =$ min else
$y = v = x$

EXAMPLE

The model line:

```
limiterint (x<-input, y->output, min<-0)
```

limits the signal `output` to positive values of the signal `input`.

NAME

`linfir` – Linear phase finite impulse response filter.

INPUT

`x`

OUTPUT

`y`

PARAMETER

`h` – A vector containing the first half of the impulse response.

`M` – Decimation ratio. Default: 1.

`skip` – Number of outputs to be suppressed from initial transient. Default: Filter order plus 1.

`isodd` – Filter has an even number of taps if `isodd` is 0, an odd number of taps if `isodd` is non-zero. Default: 0.

FUNCTION

Let N be the number of elements in the vector h. Then if `isodd` is 0:

$$y_k = \sum_{i=0}^{N-1} h_i \cdot (x_{k-i} + x_{k-2N+1+i})$$

otherwise, if `isodd` is non-zero:

$$y_k = \sum_{i=0}^{N-2} h_i \cdot (x_{k-i} + x_{k-2N+2+i}) \; + \; h_{N-1} \cdot x_{k-N+1}$$

EXAMPLE

The model lines:

```
time (k->kT, kstop<-5);
impulse (t<-kT, y->imp);
linfir (x<-imp, h<-[1,2,3], skip<-0, y->y);
monitor (x<-y);
```

prints the impulse response of the filter, $\{1,2,3,3,2,1\}$, in the output file.

BUG

Does not support filters with an anti-symmetrical impulse response (see L. Rabiner and B. Gold: "Theory and Application of Digital Signal Processing", Prentice Hall, 1975, page 82, Case 2 for a description of the filter implemented).

NAME

`linfirint` – Linear phase finite impulse response filter with integer input.

INPUT

`x` – An integer.

OUTPUT

`y`

PARAMETER

`h` – A vector containing the first half of the impulse response.
`M` – Decimation ratio. Default: 1.
`skip` – Number of outputs to be suppressed from initial transient. Default: Filter order plus 1.
`isodd` – Filter has an even number of taps if `isodd` is 0, an odd number of taps if `isodd` is non-zero. Default: 0.

FUNCTION

Let N be the number of elements in the vector h. Then if `isodd` is 0:

$$y_k = \sum_{i=0}^{N-1} h_i \cdot (x_{k-i} + x_{k-2N+1+i})$$

otherwise, if `isodd` is non-zero:

$$y_k = \sum_{i=0}^{N-2} h_i \cdot (x_{k-i} + x_{k-2N+2+i}) \; + \; h_{N-1} \cdot x_{k-N+1}$$

EXAMPLE

The model lines:

```
time (k->kT, kstop<-5);
impulse (t<-kT, y->imp);
linfir (x<-imp, h<-[1,2,3], skip<-0, y->y);
monitor (x<-y);
```

prints the impulse response of the filter, $\{1,2,3,3,2,1\}$, in the output file.

BUG

Does not support filters with an anti-symmetrical impulse response (see L. Rabiner and B. Gold: "Theory and Application of Digital Signal Processing", Prentice Hall, 1975, page 82, Case 2 for a description of the filter implemented).

NAME

`ln` – Output `y` is the natural logarithm of input `x`.

INPUT

`x`

OUTPUT

`y`

FUNCTION

$y = \ln(x)$

EXAMPLE

```
ln (x<-a, y->b);
```

BUG

The input `x` is clipped to a minimum value of 10^{-30}.

NAME

`matrix2vec` – Extract a column from a matrix.

INPUT

`m` – A matrix.

OUTPUT

`v` – A vector.

PARAMETER

`col` – Column number to be extracted. Default: 0.

FUNCTION

Output `v` is column number `col` of matrix `m`.

NAME

`matrixIn` – A clock whose output is data read from an input file.

PARAMETER

`file` – The input file.

OUTPUT

`y` – A matrix.

FUNCTION

The output pin `y` is set equal to the matrix read in from `file`. The first and second elements in the input file must be the number of columns and the number of rows in the matrix, respectively. `y` then acts as a clock signal for the netlist. Therefore, there is no need for a time base model.

EXAMPLE

The input file:

```
NETLIST
  matrixIn (file<- <data.dat>, y->nums);
  print (x1<-nums, x2<-"\n");
END
```

prints the contents of the file "`data.dat`" to the standard output as a matrix.

NAME

`max` – Output `y` is the maximum value of all inputs `x`.

INPUT

`x`

OUTPUT

`y` – Maximum of all `x`, presented at the end of the simulation.

FUNCTION

$y = \max(x(k))$

NAME

`maxinmat` – Find the largest element in a particular column of a matrix, then extract the row in which the element appears.

INPUT

`m` – A matrix.

OUTPUT

`v` – A vector.

PARAMETERS

`col` – Column in which to search for the maximum. Default: 0.
`skip` – Number of rows to skip before beginning the search. Default: 0.

FUNCTION

Output `v` is the row of `m` whose element `col` has the highest value in its column.

NAME

`maxint` – Integer output `y` is the maximum value of all integer inputs `x`.

INPUT

`x` – An integer.

OUTPUT

`y` – Maximum of all `x`, presented at the end of the simulation.

FUNCTION

$y = \max(x(k))$

NAME

`maxinvec` – Find the largest element in a vector.

INPUT

`v` – A vector.

OUTPUT

`y` – Largest element in vector `v`, presented at the end of the simulation.

FUNCTION

Output `y` is the element of `v` with the highest value.

NAME

`mean` – Output `y` is the mean of all the inputs `x` of a simulation.

INPUT

`x`

OUTPUT

`y` – Mean of all `x`, presented at the end of the simulation.

FUNCTION

$y = \frac{1}{N} \sum_i x_i$

EXAMPLE

The model lines:

```
time (k->kT, kstop<-10);
mean (x<-kT, y->Mean);
monitor (x<-Mean);
```

prints "`Mean = 5`", which is $\frac{1}{11} \sum_{i=0}^{10} i$.

BUG

No provision to obtain a running value of the mean during the simulation. Only the final result can be seen at the end of the simulation.

NAME

`min` – Output `y` is the minimum value of all inputs `x`.

INPUT

`x`

OUTPUT

`y` – Minimum of all `x`, presented at the end of the simulation.

FUNCTION

$y = \min(x(k))$

NAME

`mininmat` – Find the smallest element in a particular column of a matrix, then extract the row in which the element appears.

INPUT

`m` – A matrix.

OUTPUT

`v` – A vector.

PARAMETERS

`col` – Column in which to search for the minimum. Default: 0.
`skip` – Number of rows to skip before beginning the search. Default: 0.

FUNCTION

Output `v` is the row of `m` whose element `col` has the lowest value in its column.

NAME

`minint` – Integer output `y` is the minimum value of all integer inputs `x`.

INPUT

`x` – An integer.

OUTPUT

`y` – Minimum of all `x`, presented at the end of the simulation.

FUNCTION

$y = \min(x(k))$

NAME

`mininvec` – Find the smallest element in a vector.

INPUT

`v` – A vector.

OUTPUT

`y` – Smallest element in vector `v`, presented at the end of the simulation.

FUNCTION

Output `y` is the element of `v` with the lowest value.

NAME

`monitor` – Prints the input `x`.

INPUT

`x` – An input of any value type.

PARAMETER

`file` – The output file; the default is stdout.
`prec` – Number of significant figures to use in output. Default: 5.
`format` – Format string. Equivalent to the standard C printf format string without the leading "%". The default is the "g" format with a precision of `prec` and a length of `prec` plus six. If `format` is specified, it overrides `prec`.

FUNCTION

Prints the name of the signal connected to `x` and the value of that signal followed by a newline.

EXAMPLE

The model lines:

```
add (x1<-1, x2<-1, y->x);
monitor (x<-x, file<- <ex.dat, write>);
```

write "`x = 2`" to the file "`ex.dat`".

BUG

No checks are performed on the parameter `format`. If it is incorrectly specified, MIDAS will crash.

NAME

`mul` – Output `y` is the product of inputs `x1` and `x2`.

INPUTS

`x1`, `x2`

OUTPUT

`y`

FUNCTION

$y = x_1 \cdot x_2$

EXAMPLE

```
mul (x1<-a, x2<-b, y->c);
```

NAME

`mul4` – Output `y` is the product of up to four inputs.

INPUTS

`x1`, `x2`, `x3`, `x4` – Inputs. Default: 1.

OUTPUT

`y`

FUNCTION

$y = x_1 \cdot x_2 \cdot x_3 \cdot x_4$

EXAMPLE

The model line:

```
mul4 (x1<-3, x2<-2, x3<-.5, y->y);
```

sets the output `y` to 3.

NAME

`mul4int` – Output `y` is the integer product of up to four integers.

INPUTS

`x1`, `x2`, `x3`, `x4` – Integers. Default: 1.

OUTPUT

`y` – An integer.

FUNCTION

$y = x_1 \cdot x_2 \cdot x_3 \cdot x_4$

EXAMPLE

The model line:

```
mul4 (x1<-3, x2<-2, x3<--1, y->y);
```

sets the output `y` to -6.

NAME

`mul4intN` – Output `y` is the n-bit, 2's complement product of up to four integers.

INPUTS

`x1`, `x2`, `x3`, `x4` – Integers. Default: 1.

OUTPUT

`y` – The n-bit integer output.
`yfull` – The non-truncated integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.

PARAMETERS

`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.

FUNCTION

If the product of the inputs is greater than the maximum, $2^{n-1}-1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

BUG

The maximum value of `n` is machine dependent and is typically 31. If a greater value is specified, the model automatically reduces `n`, without warning, to the maximum supported by the platform on which the program is running.

NAME

`mulint` – Output `y` is the integer product of integers `x1` and `x2`.

INPUTS

`x1`, `x2` – Integers.

OUTPUT

`y` – An integer.

FUNCTION

$y = x_1 \cdot x_2$

NAME

`mulintN` – Output `y` is the n-bit, 2's complement product of integers `x1` and `x2`.

INPUTS

`x1`, `x2` – Integers.

OUTPUT

`y` – The n-bit integer output.
`yfull` – The non-truncated integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.

PARAMETERS

`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.

FUNCTION

If the product of the inputs is greater than the maximum, $2^{n-1}-1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

BUG

The maximum value of `n` is machine dependent and is typically 31. If a greater value is specified, the model automatically reduces `n`, without warning, to the maximum supported by the platform on which the program is running.

NAME

`mux2` – A 2-to-1 multiplexer.

INPUT

`x1`, `x2` – Inputs.
`sel` – Selects between the two inputs.

OUTPUT

`y`

FUNCTION

If $sel > 0.5$, then $y = x_1$, else $y = x_2$.

NAME

`neg` – Change the sign of the input.

INPUT

`x`

OUTPUT

`y`

FUNCTION

$y = -x$

EXAMPLE

The model file line:

```
neg (x<-5.2, y->out);
```

sets the value of `out` to -5.2.

NAME

`negint` – Change the sign of the integer input.

INPUT

`x` – An integer.

OUTPUT

`y` – An integer.

FUNCTION

$y = -x$

EXAMPLE

The model file line:

```
neg (x<--5, y->out);
```

sets the value of `out` to 5.

NAME

`Nuttall30` – A scaled window with 30dB/octave sidelobe decay. Intended for use with the `fft` and `dft` models.

INPUT

`x` – Input samples, floating point numbers.
`xint` – Input samples, integers.

OUTPUTS

`w` – A vector containing the windowed and scaled input samples, produced at the end of the simulation.
`N` – The size of `w`, i.e., the number of samples used, an integer.

PARAMETERS

`refernce` – Reference level for power: amplitude of sinusoid corresponding to 0 dB. Default: 0.5.
`DFT` – A flag indicating whether the output will be used with the `fft` or the `dft` model. If `DFT` is 0 the length of the output sequence is truncated to a power of 2, otherwise it is unaltered. Default: 0.

FUNCTION

This model first stores a sequence of simulation results as a vector of input samples. The `x` input pin should be used if the input is a real number and the `xint` input pin should be used if the input in an integer. Upon completion of the simulation, depending on the value of the `DFT` parameter, this vector is either left unchanged or truncated so as to contain the first 2^n samples, where n is an integer between 1 and a machine dependent value, typically 31. Then these samples are windowed using Nuttall's 30dB/octave window. Finally, the windowed samples are scaled such that when the FFT or DFT is taken, the result will be the spectral power of the input normalized to the power of a sinewave of amplitude given by the `reference` parameter.

EXAMPLE

The model lines:

```
Nuttall30 (x<-x, w->x_win);
fft (x<-x_win, fs<-fs, S->S);
```

```
print (x1<-S);
```

print an estimate of the spectrum of `x`, which has a sampling rate of `fs`.

REFERENCE

A. Nuttall, "Some windows with very good sidelobe behavior," IEEE Trans. on Acoustics, Speech, and Signal Processing, vol. ASSP-29, pp. 84-91, February 1981. The particular window implemented is defined in equation (33) on page 88 of this reference.

NAME

`ones` – Output a one whenever the input changes.

INPUTS

`t`

OUTPUTS

`y` – Floating point output samples equal to 1.0.
`yint` – Integer output samples equal to 1.

FUNCTION

The outputs `y` and `yint` are set equal to one and triggered whenever the input changes.

NAME

`pad` – Pad a vector with a constant.

INPUT

`x` – The input vector.

OUTPUT

`y` – The output vector.
`N` – The length of the output vector.

PARAMETERS

`num` – Number of elements to pad. Default: 0.
`val` – Value of the padded elements. Default: 0.
`DFT` – A flag indicating whether the output will be used with the `fft` or the `dft` model. If `DFT` is 0 the length of the output sequence is truncated to a power of 2, otherwise it is unaltered. Default: 0.

FUNCTION

Padding a sequence with zeros is used to increase the frequency resolution of spectral analysis.

EXAMPLE

The model lines:

```
time (k->kT, kstop<-300, kstep<-1);
sin (t<-kT, y->sq, f<-0.23456, A<-2);
Nuttall30 (x<-sq, w->nutx, reference<-2);
pad (x<-nutx, y->nut, num<-4000);
fft (x<-nut, fs<-1, S->S);
print (x1<-S, x2<-"\n");
```

produce a 4096-point estimate of the input spectrum, 2048 points of which are printed out by MIDAS. See the `fft` model description for more information regarding the output.

NAME

`pdf` – Computes the probability density function for the signal `x`.

INPUT

`x` – The input samples.
`xint` – The input samples, integers.

OUTPUTS

`pdf` – A matrix: column 0 is scale (middle of interval), column 1 is pdf.

PARAMETERS

`min` – Lower end of range of pdf. Default: -1.0.
`max` – Upper end of range of pdf. Default: $+1.0$.
`resolution` – Number of intervals (bins) for estimation. Default: 100.

FUNCTION

The inputs are placed in bins according to where they fit in the interval range. The `x` input pin should be used if the inputs are real numbers and the `xint` input pin should be used if the inputs are integers. The bins are then normalized so that the sum of the bins multiplied by the bin width is one. This approximates a probability density function.

NAME

`pow` – Output `y` is the `a`th power of input `x`.

INPUT

`x`
`a` – Default: 1.

OUTPUT

`y`

FUNCTION

$y = x^a$

EXAMPLE

```
pow (x<-x, a<-4, y->y);
```

NAME

`power` – Output `y` is the average power of all the inputs `x` in a simulation.

INPUT

`x`

OUTPUT

`y` – Average power of all `x`, presented at the end of the simulation.
`dB` – Average power of all `x` in dB, presented at the end of the simulation. (0dB corresponds to an average power of 1.)

FUNCTION

$y = \frac{1}{N} \sum_i x_i^2$

EXAMPLE

The model lines:

```
time (k->kT, kstop<-5);
power (x<-kT, y->Power);
monitor (x<-Power);
```

prints "`Power = 9.167`", which is $\frac{1}{6} \sum_{i=0}^{5} i^2$.

BUG

No provision to obtain a running value of the power during the simulation. Only the final result can be seen at the end of the simulation.

NAME

`powint` – Integer output `y` is the `a`th power of integer input `x`.

INPUT

`x` – An integer.
`a` – An integer. Default: 1.

OUTPUT

`y` – An integer.

FUNCTION

$y = x^a$

EXAMPLE

```
powint (x<-x, a<-4, y->y);
```

NAME

`print` – Prints its arguments.

INPUTS

`xi` – A signal of any type. The `i` in `xi` can be blank or any digit from 1 to 9.
`trigger` – Synchronize printing with change of this signal.

PARAMETERS

`file` – The output file. Default: `stdout`.
`prec` – Number of significant figures to use in output. Default: 5.
`format` – Format string. Equivalent to the standard C printf format string without the leading "%". The default is the "g" format with a precision of `prec` and a length of `prec` plus six. If `format` is specified, it overrides `prec`.
`transpose` – If `transpose` is 0, vectors are printed as row vectors and matrices are printed row-wise. If `transpose` is not 0, vectors are printed as column vectors and matrices are printed column-wise. Default: 0.

FUNCTION

Prints each argument in order x, x1, x2, ... This model always prints all of its arguments at once. By default, printing occurs when at least one input changes. Alternatively, a trigger signal can be specified.

EXAMPLE

The model lines:

```
add (x1<-1, x2<-1, y->y);
print (x1<-"Y = ", x2<-y, x3<-" dB\n");
```

prints "`Y = 2 dB`", followed by a newline.

BUG

No checks are performed on the parameter `format`. If it is incorrectly specified, MIDAS will crash.

NAME

`pulse` – Pulse train signal source.

INPUT

`x` – Triggers the output.

OUTPUTS

`y`
`yint` – An integer.

PARAMETERS

`M` – Number of clock samples in pulse period. Default: 1.
`W` – Number of clock samples in pulse width. Default: 1.
`D` – Number of clock samples of delay before the first period starts. Default: 0.
`scale` – Output value when the output is active. Default: 1.

FUNCTION

The output `y` is 0 for D samples. The clock period then begins with the output equal to *scale* for W samples and equal to 0 for $M - W$ samples. The output is periodic thereafter.

EXAMPLE

The model line:

```
time (t->kT, kstep<-1, kstop<-14);
pulse (x<-kT, y->out, M<-3, W<-2, D<-4, scale<-2.2);
```

produces the following sequence:

```
0 0 0 0 2.2 2.2 0 2.2 2.2 0 2.2 2.2 0 2.2 2.2
```

NAME

`pureDelay` – Output `y` is `x1` delayed by one clock cycle.

INPUT

`x1`

OUTPUTS

`y` – Delayed (late) output.
`v` – Immediate output.

PARAMETER

`x0` – Initial state ($y[0]$). Default: 0.

FUNCTION

$y[k+1] = x1[k]$
$v[k] = x1[k]$

EXAMPLE

```
pureDelay (x1<-input, y->output);
```

NAME

`quant` – Two level quantizer.

INPUT

`x`

OUTPUT

`y`

PARAMETERS

`yp` – Output level for positive input. Default: +0.5.
`ym` – Output level for negative input. Default: −0.5.
`hysteresis` – Default: 0.

FUNCTION

if ($\|x\| < hysteresis/2$)	then	$y =$ last y		
	else	if ($x > 0$)	then	$y = yp$
			else	$y = ym$

The initial "last y" is defined to be yp.

NAME

`quantint` – Two level quantizer with integer input and output.

INPUT

`x` – Input signal, an integer.

OUTPUT

`y` – Output signal, an integer.

PARAMETERS

`yp` – Output level for positive input. Default: +1.
`ym` – Output level for negative input. Default: −1.
`hysteresis` – Default: 0.

FUNCTION

if ($|x| < hysteresis/2$) then $y =$ last y
else if ($x > 0$) then $y = yp$
else $y = ym$

The initial "last y" is defined to be yp.

NAME

`quantizer` – Arbitrary shape quantizer.

INPUT

`x`

OUTPUT

`y`

PARAMETERS

`xx` – A vector whose elements are the input bin boundaries.
`yy` – A vector whose elements are the ouput values (dimension: 1 longer than `xx`).

FUNCTION

if $x < x_0$ then $y = y_0$ else
if $x < x_1$ then $y = y_1$ else
...
if $x < x_n$ then $y = y_n$ else
$y = y_{n+1}$

BUG

Values of `xx` must rise monotonically for the output `y` to make sense.

SEE ALSO

`quantUniform`

NAME

`quantizerint` – Arbitrary shape quantizer with integer input and output.

INPUT

`x` – An integer.

OUTPUT

`y` – An integer.

PARAMETERS

`xx` – A vector whose elements are the input bin boundaries.
`yy` – A vector whose elements are the ouput values (dimension: 1 longer than `xx`).

FUNCTION

if $x < x_0$ then $y = y_0$ else
if $x < x_1$ then $y = y_1$ else
...
if $x < x_n$ then $y = y_n$ else
$y = y_{n+1}$

BUG

Values of `xx` must rise monotonically for the output `y` to make sense.

SEE ALSO

`quantUniform`

NAME

`quantnlave` – A quantizer composed of an ideal ADC and a DAC with random mismatches. The polarity of the mismatches alternates every time a level is reached.

INPUT

`x` – Input signal.

OUTPUT

`y` – Output signal.
`levels` – Vector containing the output levels. Available at the end of the simulation.

PARAMETERS

`bottomRail` – Lowest output signal level. Default: -0.5.
`topRail` – Highest output signal level. Default: +0.5.
`numBins` – Number of bins in the quantizer. Default: 2.
`linearity` – Standard deviation of mismatch in LSB's. Default: 0.
`seed` – Seed for radom number generator. Default: 1.

FUNCTION

This model is similar to the `quantnldac` model except that the polarity of the error at each quantizer output level is toggled everytime that level is invoked.

SEE ALSO

`quantnldac`, `quantUniform`

NAME

`quantnldac` – A quantizer composed of an ideal ADC and a DAC with random mismatches.

INPUT

`x` – Input signal.

OUTPUT

`y` – Output signal.
`levels` – Vector containing the output levels. Available at the end of the simulation.

PARAMETERS

`bottomRail` – Lowest output signal level. Default: -0.5.
`topRail` – Highest output signal level. Default: +0.5.
`numBins` – Number of bins in the quantizer. Default: 2.
`linearity` – Standard deviation of mismatch in LSB's. Default: 0.
`seed` – Seed for radom number generator. Default: 1.

FUNCTION

This model is similar to the `quantUniform` model except that when the output levels are computed, a Gaussian random number is added to each of the ideal levels. These output levels are then used throught the simulation. The mean of the random variable is 0 and its standard deviation is equal to the quantizer LSB multiplied by the linearity parameter, as shown below:

$$\sigma = \frac{topRail - bottomRail}{numBins - 1} \cdot linearity$$

SEE ALSO

`quantUniform`

NAME

`quantUniform` – Uniform quantizer.

INPUT

`x` – Input signal.

OUTPUT

`y` – Output signal.

PARAMETERS

`bottomRail` – Lowest output signal level. Default: -0.5.
`topRail` – Highest output signal level. Default: +0.5.
`numBins` – Number of bins in the quantizer. Default: 2.

FUNCTION

Let:

$$binSize = \frac{topRail - bottomRail}{numBins - 1}.$$

Then for inputs within the two rails:

$$y = \text{int}\left(\frac{x - bottomRail}{binSize}\right) \cdot binSize + bottomRail$$

where the "int" function rounds to the nearest integer. If the input is outside the two rails, it is clipped at the nearest rail.

NAME

`quantUniformint` – Uniform quantizer with integer input and output.

INPUT

`x` – Input signal, an integer.

OUTPUT

`y` – Output signal, an integer.

PARAMETERS

`bottomRail` – Lowest output signal level. Default: -1.
`topRail` – Highest output signal level. Default: +1.
`numBins` – Number of bins in the quantizer. Default: 2.

FUNCTION

Let:

$$binSize = \frac{topRail - bottomRail}{numBins - 1}.$$

Then for inputs within the two rails:

$$y = \text{int}\left(\frac{x - bottomRail}{binSize}\right) \cdot binSize + bottomRail$$

where the "int" function rounds to the nearest integer. If the input is outside the two rails, it is clipped at the nearest rail.

NAME

`rectangular` – A scaled rectangular window. Intended for use with the `fft` and `dft` models. This model is included for completeness; it is almost always better to use a different windowing function.

INPUT

`x` – Input samples, floating point numbers.
`xint` – Input samples, integers.

OUTPUTS

`w` – A vector containing the windowed and scaled input samples, produced at the end of the simulation.
`N` – The size of `w`, i.e., the number of samples used, an integer.

PARAMETERS

`refernce` – Reference level for power: amplitude of sinusoid corresponding to 0 dB. Default: 0.5.
`DFT` – A flag indicating whether the output will be used with the `fft` or the `dft` model. If `DFT` is 0 the length of the output sequence is truncated to a power of 2, otherwise it is unaltered. Default: 0.

FUNCTION

This model first stores a sequence of simulation results as a vector of input samples. The `x` input pin should be used if the input is a real number and the `xint` input pin should be used if the input in an integer. Upon completion of the simulation, depending on the value of the `DFT` parameter, this vector is either left unchanged or truncated so as to contain the first 2^n samples, where n is an integer between 1 and a machine dependent value, typically 31. Then these samples are windowed using the rectangular window. Finally, the windowed samples are scaled such that when the FFT or DFT is taken, the result will be the spectral power of the input normalized to the power of a sinewave of amplitude given by the `reference` parameter.

EXAMPLE

The model lines:

```
rectangular (x<-x, w->x_win);
```

```
fft (x<-x_win, fs<-fs, S->S);
print (x1<-S);
```

print an estimate of the spectrum of `x`, which has a sampling rate of `fs`.

SEE ALSO

`Nuttall30`

REFERENCE

A. Oppenheim and R. Schafer, "Discrete-Time Signal Processing," Prentice Hall, 1989.

NAME

`settle` – Slew rate limited exponential response model.

INPUT

`x` – Input samples.

OUTPUT

`y` – `x` after passing though the slew limiting and exponential response element.

PARAMETERS

`T` – Total response time, in seconds. Default: 100^{-9}.
`tau` – Exponential time constant, in seconds. Default: 10^{-9}.
`s` – Slew rate, in volts per second. Default: 100^{+6}.

FUNCTION

Device has response $y = x(1 - e^{-T/\tau})$ with the rate of change for entire exponential limited to s/T. T is the time available for the integrator response and is typically one half the sampling period.

EXAMPLE

The model lines:

```
mul (x1<-input, x2<-0.5, y->hlf);
settle (x<-hlf, y->response, T<-60e-9, tau<-10e-9, s<-7e6);
delay (x1<-response, x2<-output, y->output)
```

create an integrator with gain of 0.5, settling time constant of $10ns$, and slew rate of $7V/\mu s$ in a $60ns$ integration phase.

BUGS

Since the exponential portion of the response is modelled as perfectly linear, analysis with this model suggests that a high resolution sigma-delta modulator can be constructed with low bandwidth and high slew rate integrators. In practice, exponential response is never perfectly linear and high resolution data conversion is difficult to achieve with low bandwidth integrators.

NAME

`shift` – Perform a binary shift on an integer.

INPUT

`x`

OUTPUT

`y`

PARAMETER

`n` – Number of positions to shift the input. A negative `n` results in a right shift and a positive `n` results in a left shift. Default: 0.

FUNCTION

A left shift is equivalent to multiply by 2 and a right shift is equivalent to divide by 2 where the remainder is discarded.

NAME

`sigma1` – A first order sigma-delta modulator.

INPUT

`x` – Input of modulator.

OUTPUTS

`y` – Output of quantizer.
`q` – Output of D/A; equal to `G` times `y`.
`u` – Output of integrator; at the input of the quantizer.

PARAMETERS

`a` – Gain of integrator. Default: 1.
`G` – Gain of D/A. Default: 1.
`u0` – Initial value of `u`. Default: 0.

FUNCTION

This model is equivalent to the model lines:

```
sub (x1<-x, x2<-q, y->internal1);
delay (x1<-internal1, x2<-u, y->u, c1<-a, x0<-u0);
quant (x<-u, y->y);
mul (x1<-y, x2<-G, y->q);
```

NAME

`sigma2` – A second order sigma-delta modulator.

INPUT

`x` – Input of modulator.

OUTPUTS

`y` – Output of quantizer.
`q` – Output of D/A; equal to `G` times `y`.
`u1` – Output of first integrator.
`u2` – Output of second integrator.

PARAMETERS

`a1` – Gain of first integrator.
`a2` – Gain of second integrator.
`G` – Gain of D/A.
`u1_0` – Initial value of `u1`.
`u2_0` – Initial value of `u2`.

FUNCTION

This model is equivalent to the model lines:

```
sub (x1<-x, x2<-q, y->internal1);
delay (x1<-internal1, x2<-u1, y->u1, c1<-a1, x0<-u1_0);
sub (x1<-u1, x2<-q, y->internal2);
delay (x1<-internal2, x2<-u2, y->u2, c1<-a2, x0<-u2_0);
quant (x<-u2, y->y);
mul (x1<-y, x2<-G, y->q);
```

NAME

`sin` – Sampled sinusoidal waveform generator.

INPUT

`t` – Time.

OUTPUT

`y`

PARAMETERS

`A` – Amplitude of sinusoid with gain at 0dB. Default: 1.0.
`gain` – Modify amplitude of sinusoid by `gain` dB. Default: 0.
`f` – Frequency. Default: 1.0.

FUNCTION

$y = A \cdot 10^{gain/20} \cdot \sin(2\pi f t)$

EXAMPLE

The model lines:

```
sin (t<-1/4000, A<-5, f<-1000, y->Out);
monitor (x<-Out);
```

print "`Out = 5.000`".

NAME

`sinc3` – Linear phase comb-type decimation filter.

INPUT

`x`

OUTPUT

`y`

PARAMETERS

`M` – Decimation ratio, ≥ 1. Default: 1.
`skip` – Number of outputs from initial transient to be suppressed. Default: 3 (i.e. skip entire initial transient).

FUNCTION

Implements the following filter transfer function:

$$H(z) = \left(\frac{1}{M} \sum_{k=0}^{M-1} z^{-k} \right)^3$$

The output `y` is `x` filtered and decimated by the factor `M`.

EXAMPLE

The model lines:

```
time (k->kT, kstop<-20);
impulse (t<-kT, y->x);
sinc3 (x<-x, y->y, skip<-0, M<-8);
monitor (x<-y);
```

print:

```
y = 0.070313
y = 0.054688
```

to stdout.

NAME

`sinc3int` – Linear phase comb-type decimation filter with integer input.

INPUT

`x` – An integer.

OUTPUT

`y`

PARAMETERS

`M` – Decimation ratio, ≥ 1. Default: 1.
`skip` – Number of outputs from initial transient to be suppressed. Default: 3 (i.e. skip entire initial transient).

FUNCTION

Implements the following filter transfer function:

$$H(z) = \left(\frac{1}{M} \sum_{k=0}^{M-1} z^{-k} \right)^3$$

The output `y` is `x` filtered and decimated by the factor `M`.

EXAMPLE

The model lines:

```
time (k->kT, kstop<-20);
impulse (t<-kT, y->x);
sinc3 (x<-x, y->y, skip<-0, M<-8);
monitor (x<-y);
```

print:

```
y = 0.070313
y = 0.054688
```

to stdout.

NAME

`sinc4` – Linear phase comb-type decimation filter.

INPUT

`x`

OUTPUT

`y`

PARAMETER

`M` – Decimation ratio, ≥ 1. Default: 1.
`skip` – Number of outputs from initial transient to be suppressed. Default: 4 (i.e. skip entire initial transient).

FUNCTION

Implements the following filter transfer function:

$$H(z) = \left(\frac{1}{M} \sum_{k=0}^{M-1} z^{-k} \right)^4$$

The output `y` is `x` filtered and decimated by the factor `M`.

EXAMPLE

The model lines:

```
time (k->kT, kstop<-20);
impulse (t<-kT, y->x);
sinc4 (x<-x, y->y, skip<-0, M<-8);
monitor (x<-y);
```

print:

```
y = 0.029297
y = 0.082031
```

to stdout.

NAME

`sinc4int` – Linear phase comb-type decimation filter with integer input.

INPUT

`x` – An integer.

OUTPUT

`y`

PARAMETER

`M` – Decimation ratio, ≥ 1. Default: 1.
`skip` – Number of outputs from initial transient to be suppressed. Default: 4 (i.e. skip entire initial transient).

FUNCTION

Implements the following filter transfer function:

$$H(z) = \left(\frac{1}{M} \sum_{k=0}^{M-1} z^{-k} \right)^4$$

The output `y` is `x` filtered and decimated by the factor `M`.

EXAMPLE

The model lines:

```
time (k->kT, kstop<-20);
impulse (t<-kT, y->x);
sinc4 (x<-x, y->y, skip<-0, M<-8);
monitor (x<-y);
```

print:

```
y = 0.029297
y = 0.082031
```

to stdout.

NAME

`sincN` – Linear phase comb-type decimation filter.

INPUT

`x`

OUTPUT

`y`

PARAMETERS

`M` – Decimation ratio, ≥ 1. Default: 1.
`N` – Filter order. Default: 1.
`skip` – Number of outputs from initial transient to be suppressed. Default: N (i.e. skip entire initial transient).

FUNCTION

Implements the following filter transfer function:

$$H(z) = \left(\frac{1}{M} \sum_{k=0}^{M-1} z^{-k} \right)^N$$

The output `y` is `x` filtered and decimated by factor M.

EXAMPLE

The model lines:

```
time (k->kT, kstop<-20);
impulse (t<-kT, y->x);
sincN (x<-x, y->y, skip<-0, M<-8, N<-3);
monitor (x<-y);
```

print:

```
y = 0.070313
y = 0.054688
```

to stdout.

NAME

`sincNint` – Linear phase comb-type decimation filter with integer input.

INPUT

`x` – An integer.

OUTPUT

`y`

PARAMETERS

`M` – Decimation ratio, ≥ 1. Default: 1.
`N` – Filter order. Default: 1.
`skip` – Number of outputs from initial transient to be suppressed. Default: N (i.e. skip entire initial transient).

FUNCTION

Implements the following filter transfer function:

$$H(z) = \left(\frac{1}{M} \sum_{k=0}^{M-1} z^{-k} \right)^N$$

The output `y` is `x` filtered and decimated by factor M.

EXAMPLE

The model lines:

```
time (k->kT, kstop<-20);
impulse (t<-kT, y->x);
sincN (x<-x, y->y, skip<-0, M<-8, N<-3);
monitor (x<-y);
```

print:

```
y = 0.070313
y = 0.054688
```

to stdout.

NAME

`sinN` – Sampled sinusoidal waveform generator. The output is an n-bit, 2's complement integer.

INPUT

`t` – Time, a floating point number.

OUTPUT

`y` – The n-bit integer output.
`yfull` – The non-truncated integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.

PARAMETERS

`A` – Amplitude of sinusoid with gain at 0dB. Default: 1.0.
`gain` – Modify amplitude of sinusoid by `gain` dB. Default: 0.
`f` – Frequency. Default: 1.0.
`phase` – Phase in radians. Default: 0.0.
`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.

FUNCTION

The output is first computed according to the following equation:

$$y = A \cdot 10^{gain/20} \cdot \sin(2\pi ft + phase)$$

and converted to an integer. Then, if `y` is greater than the maximum, $2^{n-1} - 1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

BUG

The maximum value of `n` is machine dependent and is typically 31. If a greater value is specified, the model automatically reduces `n`, without warning, to the maximum supported by the platform on which the program is running.

NAME

`sinNzins` – Sampled sinusoidal waveform generator with zero insertion. Output `y` is the n-bit, 2's complement sum of integer inputs `x1` and `x2`.

INPUT

`t` – Time.

OUTPUT

`y` – The n-bit integer output.
`yfull` – The non-truncated integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.

PARAMETERS

`A` – Amplitude of sinusoid with gain at 0dB. Default: 1.0.
`gain` – Modify amplitude of sinusoid by `gain` dB. Default: 0.
`f` – Frequency. Default: 1.0.
`phase` – Phase in radians. Default: 0.0.
`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.
`M` – In every `M` output samples `M`-1 zeros are inserted. Default: 1.0.

FUNCTION

At the first input sample and at every `M` samples after that, the output is generated according to:

$$y = A \cdot 10^{gain/20} \cdot \sin(2\pi ft + phase)$$

and converted to an integer. For all other samples, the output is equal to 0. Then, if `y` is greater than the maximum, $2^{n-1} - 1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

SEE ALSO

`sinzins`, `sinzoh`, `sinNzoh`

NAME

`sinNzoh` – Sampled sinusoidal waveform generator with zero insertion. Output `y` is the n-bit, 2's complement sum of integer inputs `x1` and `x2`.

INPUT

`t` – Time.

OUTPUT

`y` – The n-bit integer output.
`yfull` – The non-truncated integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.

PARAMETERS

`A` – Amplitude of sinusoid with gain at 0dB. Default: 1.0.
`gain` – Modify amplitude of sinusoid by `gain` dB. Default: 0.
`f` – Frequency. Default: 1.0.
`phase` – Phase in radians. Default: 0.0.
`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.
`M` – Output is held constant for `M` samples. Default: 1.0.

FUNCTION

At the first input sample and at every `M` samples after that, the output is generated according to:

$$y = A \cdot 10^{gain/20} \cdot \sin(2\pi ft + phase)$$

and converted to an integer. For all other samples, the output is held constant at its previous value. Then, if `y` is greater than the maximum, $2^{n-1} - 1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

SEE ALSO

`sinzins`, `sinzoh`, `sinNzins`

NAME

`sinzins` – Sampled sinusoidal waveform generator with zero insertion.

INPUT

`t` – Time.

OUTPUT

`y`

PARAMETERS

`A` – Amplitude of sinusoid with gain at 0dB. Default: 1.0.
`gain` – Modify amplitude of sinusoid by `gain` dB. Default: 0.
`f` – Frequency. Default: 1.0.
`phase` – Phase in radians. Default: 0.0.
`M` – In every `M` output samples `M`-1 zeros are inserted. Default: 1.0.

FUNCTION

At the first input sample and at every `M` samples after that, the output is generated according to:

$$y = A \cdot 10^{gain/20} \cdot \sin(2\pi f t + phase).$$

For all other samples, the output is equal to 0.

EXAMPLE

The model lines:

```
time (k->kT, kstart<-0.1, kstep<-0.11, kstop<-0.9);
sinzins (t<-kT, y->y, M<-3);
monitor (x<-y);
```

print:

```
y =     0.58779
y =           0
y =           0
y =     0.42578
y =           0
y =           0
```

```
y =    -0.99803
y =           0
```

to stdout.

SEE ALSO

`sinzins2comp`, `sinzoh`, `sinzoh2comp`

NAME

`sinzoh` – Sampled sinusoidal waveform generator with zero-order hold.

INPUT

`t` – Time.

OUTPUT

`y`

PARAMETERS

`A` – Amplitude of sinusoid with gain at 0dB. Default: 1.0.
`gain` – Modify amplitude of sinusoid by `gain` dB. Default: 0.
`f` – Frequency. Default: 1.0.
`phase` – Phase in radians. Default: 0.0.
`M` – Output is held constant for `M` samples. Default: 1.0.

FUNCTION

At the first input sample and at every `M` samples after that, the output is generated according to:

$$y = A \cdot 10^{gain/20} \cdot \sin(2\pi f t + phase).$$

For all other samples, the output is held constant at its previous value.

EXAMPLE

The model lines:

```
time (k->kT, kstart<-0.1, kstep<-0.11, kstop<-0.9);
sinzins (t<-kT, y->y, M<-3);
monitor (x<-y);
```

print:

```
y =     0.58779
y =     0.58779
y =     0.58779
y =     0.42578
y =     0.42578
y =     0.42578
```

```
y =    -0.99803
y =    -0.99803
```

to stdout.

SEE ALSO

`sinzins`, `sinzoh2comp`, `sinzoh2comp`

NAME

`skip` – Skip initial transient.

INPUT

`x`

OUTPUT

`y`

PARAMETER

`skip` – Number of outputs from initial transient to be suppressed. Default: 1.

NAME

`skipint` – Skip initial transient for integers.

INPUT

`x` – An integer.

OUTPUT

`y` – An integer.

PARAMETER

`skip` – Number of outputs from initial transient to be suppressed. Default: 1.

NAME

`sqrt` – Output `y` is the square root of the input `x`.

INPUT

`x`

OUTPUT

`y`

FUNCTION

$y = \sqrt{|x|}$

EXAMPLE

```
sqrt (x<-a, y->b);
```

NAME

`sub` – Output `y` is the difference of the inputs `x1` and `x2`.

INPUTS

`x1`, `x2`

OUTPUT

`y`

FUNCTION

$y = x1 - x2$

EXAMPLE

```
sub (x1<-a, x2<-b, y->c);
```

NAME

`subint` – Output `y` is the integer difference of integer inputs `x1` and `x2`.

INPUTS

`x1`, `x2` – Integers.

OUTPUT

`y` – An integer.

FUNCTION

$y = x1 - x2$

EXAMPLE

```
subint (x1<-a, x2<-b, y->c);
```

NAME

`subintN` – Output `y` is the n-bit, 2's complement difference of integer inputs `x1` and `x2`.

INPUTS

`x1`, `x2` – Integers.

OUTPUT

`y` – The n-bit integer output.
`yfull` – The non-truncated integer output.
`overflow` – Number of times an overflow has occurred.
`underflow` – Number of times an underflow has occurred.

PARAMETERS

`n` – Number of bits with which the output is represented. Default: 31.
`clip` – A flag indicating whether the output should change during an overflow or underflow or remain clipped at the extremum. If `clip` is 0, the output is clipped, otherwise it wraps around. Default: 0.

FUNCTION

If the difference of the inputs is greater than the maximum, $2^{n-1} - 1$, or less than the minimum, -2^{n-1}, allowed by an n-bit, 2's complement representation, `y` is either clipped at the extermum or allowed to wrap around in a 2's complement fashion, depending on the value of the flag `clip`. Everytime an overflow or underflow occurs, the `overflow` or the `underflow` output is incremented, respectively.

BUG

The maximum value of `n` is machine dependent and is typically 31. If a greater value is specified, the model automatically reduces `n`, without warning, to the maximum supported by the platform on which the program is running.

NAME

`tapDelay` – A tapped delay line filter.

INPUT

`x`

OUTPUTS

`y` – Filter output.
`x1`, `x2`, `x3`, `x4` – Delayed versions of `x`.

PARAMETERS

`a0`, `a1`, `a2`, `a3`, `a4` – Filter parameters. Default: 0.
`x1_0`, `x2_0`, `x3_0`, `x4_0` – Initial values for `x1` - `x4`. Default: 0.

FUNCTION

$y = x \cdot (a_0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3} + a_4 z^{-4})$
$x_1 = x z^{-1}$
$x_2 = x z^{-2}$
$x_3 = x z^{-3}$
$x_4 = x z^{-4}$

EXAMPLE

The model line:

```
tapDelay (a0<-1, a1<-(-2), a2<-1, x<-x, y->y);
```

creates the function:

$$y = (1 - z^{-1})^2$$

NAME

`tapDelayint` – A tapped delay line filter with integer input and output.

INPUT

`x` – An integer.

OUTPUTS

`y` – Filter output, an integer.
`x1`, `x2`, `x3`, `x4` – Delayed versions of `x`.

PARAMETERS

`a0`, `a1`, `a2`, `a3`, `a4` – Filter parameters. Default: 0.
`x1_0`, `x2_0`, `x3_0`, `x4_0` – Initial values for `x1` - `x4`. Default: 0.

FUNCTION

$y = x \cdot (a_0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3} + a_4 z^{-4})$
$x_1 = x z^{-1}$
$x_2 = x z^{-2}$
$x_3 = x z^{-3}$
$x_4 = x z^{-4}$

EXAMPLE

The model line:

```
tapDelayint (a0<-1, a1<-(-2), a2<-1, x<-x, y->y);
```

creates the function:

$$y = (1 - z^{-1})^2$$

NAME

`time` – Discrete time generator.

PARAMETERS

`kstart` – Initial value of `k`. Default: 0.
`kstop` – Final value of `k`. Default: 0.
`kstep` – Time interval. Default: 1.0.

OUTPUT

`k` – Time step.

FUNCTION

`time` is called during every simulation cycle and generates the sequence:

$k = kstart$
$k = kstart + kstep$
$k = kstart + 2\,kstep$
...
$k \geq kstop$

EXAMPLE

The model lines:

```
time (k->kT, kstep<-0.5, kstop<-2.5);
monitor (x<-kT);
```

print the sequence:

```
kT = 0.000
kT = 0.500
kT = 1.000
kT = 1.500
kT = 2.000
kT = 2.500
```

NAME

`timeint` – Discrete time generator with integer outputs.

PARAMETERS

`kstart` – Initial value of `k`. Default: 0.
`kstop` – Final value of `k`. Default: 0.
`kstep` – Time interval. Default: 1.0.

OUTPUT

`k` – Time step, an integer.

FUNCTION

`time` is called during every simulation cycle and generates the sequence:

$k = kstart$
$k = kstart + kstep$
$k = kstart + 2\,kstep$
...
$k \geq kstop$

EXAMPLE

The model lines:

```
timeint (k->kT, kstep<-1, kstop<-4);
monitor (x<-kT);
```

print the sequence:

```
kT = 0
kT = 1
kT = 2
kT = 3
kT = 4
```

NAME

`uniform` – Uniform random number generator.

INPUT

`t` – Clock (a random number is generated for every occurrance of t).

OUTPUT

`y`

PARAMETERS

`mean` – Mean of the random numbers. Default: 0.
`range` – Range of the random numbers (largest minus smallest). Default: 1.
`seed` – Seed for random number generator. Default: 1.

FUNCTION

Generates random numbers with uniform distribution in interval:

$$mean - \frac{range}{2} \cdots mean + \frac{range}{2}$$

EXAMPLE

The model lines:

```
time (k->kT, kstart<-1, kstop<-10);
uniform (t<-kT, y->y);
monitor (x<-y);
```

print out ten random numbers in interval ± 0.5.

BUG

This model is based on the standard UNIX function "`random.c`"; results might be incorrect on other systems. See the UNIX manual for the properties of "`random`".

NAME

`variance` – Output `y` is the variance of all the inputs `x` in a simulation.

INPUT

`x`

OUTPUT

`y` – Variance of all `x`, presented at the end of the simulation.
`dB` – Variance of all `x` in dB, presented at the end of the simulation. (0dB corresponds to a variance of 1.)

FUNCTION

$y = \frac{1}{N} \sum_i x_i^2 - \frac{1}{N} \sum_i x_i$

NAME

`vectorIn` – A clock whose output is data read from an input file.

PARAMETER

`file` – The input file.

OUTPUT

`y` – A vector.

FUNCTION

The output pin `y` is set equal to the vector read in from `file`. The length of the vector need not be specified explicitly. `y` then acts as a clock signal for the netlist. Therefore, there is no need for a time base model.

EXAMPLE

The input file:

```
NETLIST
  vectorIn (file<- <data.dat>, y->nums);
  print (x1<-nums, x2<-"\n");
END
```

prints the contents of the file "`data.dat`" to the standard output as a vector.

NAME

`zeros` – Output a zero whenever the input changes.

INPUTS

`t`

OUTPUTS

`y` – Floating point output samples equal to 0.0.
`yint` – Integer output samples equal to 0.

FUNCTION

The outputs `y` and `yint` are set equal to zero and triggered whenever the input changes.

NAME

`zins` – Zero insertion.

INPUT

`x` – Input.
`clk` – Trigger.

OUTPUT

`y`

PARAMETERS

`M` – In every `M` output samples `M`-1 zeros are inserted. Default: 1.0.

FUNCTION

Output `y` is triggered every cycle that `x` or `clk` change but is set equal to the input every M cycles and is equal to zero otherwise.

NAME

`zinsint` – Zero insertion with integer input and output.

INPUT

`x` – Input, an integer.
`clk` – Trigger, an integer.

OUTPUT

`y` – An integer.

PARAMETERS

`M` – In every `M` output samples `M`-1 zeros are inserted. Default: 1.0.

FUNCTION

Output `y` is triggered every cycle that `x` or `clk` change but is set equal to the input every M cycles and is equal to zero otherwise.

NAME

`zoh` – Zero-order hold.

INPUT

`x` – Input.
`clk` – Trigger.

OUTPUT

`y`

PARAMETERS

`M` – Output is held constant for `M` samples. Default: 1.0.

FUNCTION

Output `y` is triggered every cycle that `x` or `clk` change but is set equal to the input every M cycles and is held constant at its previous value otherwise.

NAME

`zohint` – Zero-order hold with integer input and output.

INPUT

`x` – Input, an integer.
`clk` – Trigger, an integer.

OUTPUT

`y` – An integer.

PARAMETERS

`M` – Output is held constant for `M` samples. Default: 1.0.

FUNCTION

Output `y` is triggered every cycle that `x` or `clk` change but is set equal to the input every M cycles and is held constant at its previous value otherwise.