

# **IBIS Macro Language Reference Manual**

I/O Buffer Information Specification Macro Language (IBIS-ML)  
Language Reference Manual Version 0.3c.

The IBIS Macro Language is a standard language suitable for modeling the analog characteristics of a digital integrated circuit. The IBIS Macro Language is intended for use in conjunction with the EIA IBIS standard.

---



---

TABLE OF CONTENTS

---



---

<b>1. THE IBIS - X MACRO LANGUAGE .....</b>	<b>5</b>
<b>2. SYNTAX RULES AND GUIDELINES.....</b>	<b>6</b>
2.1 Syntax Rules.....	6
2.2 Reserved Words and Pre-defined Constants .....	7
<b>3. THE [ DEFINE ] KEYWORD.....</b>	<b>8</b>
<b>4. GENERAL OVERVIEW.....</b>	<b>9</b>
4.1 Introduction .....	9
4.2: A Buffer Example .....	9
4.3 IBIS-ML Library File.....	11
<b>5. BASIC CIRCUIT ELEMENTS.....</b>	<b>13</b>
5.1 Basic Circuit Components.....	13
5.2 Basic Components.....	14
5.3 Instantiating Basic Circuit Components.....	15
5.4 Assigning Values To A component .....	17
5.4 Instantiating a Subcircuit.....	18
<b>6. EXPRESSIONS, FUNCTIONS, TABLES, AND MATRICES.....</b>	<b>19</b>
6.1 Symbolic Values .....	19
6.2 Expressions.....	19
6.3 Tables .....	20
6.4 Matrices.....	20
6.5 Functions .....	20
<b>7. THE DEFINE CONSTRUCT.....</b>	<b>21</b>
<b>8. TEST AND CONTROL CONSTRUCTS.....</b>	<b>22</b>
8.1 The "assert" Statement .....	23
8.2 The "define" Statement .....	24
8.3 The "foreach" Statement .....	25
8.4 The "if" Statement.....	27
8.5 The "inherit" Statement.....	29
8.6 The "node" Statement.....	30
8.7 The "select" Control Statement .....	31
<b>9. DATA TYPES.....</b>	<b>35</b>

9.1 Strings ..... 35

9.2 Scalars ..... 36

9.3 Tables ..... 37

9.4 Multi-dimensional tables..... 39

9.5 Scopes..... 41

9.6 Foreach arrays ..... 43

10. FUTURE EXTENSIONS..... 45

temporary use only..... 46

---

---

REVISION HISTORY

---

---

10/9/01 – Lynne Green

- Rev 0.3c
- Added format for tlines.
- Cleanup of Sections 2, 4-5.2, based on committee meeting of Oct 4.
- Put Section 3: The Define Keyword, back in.
- Turned autonumbering on for the sections. Regenerated TOC.

09/7/01 – Lynne Green

- Rev 0.3b
- Removed references to IBIS-X specifically, replaced with reference to IBIS or EIA/IBIS specification.
- Complete rewrite of Section 2, based on Stephen's draft & committee comments.
- Rewrite of section 3.1.
- Rewrite of section 3.2 (first model example).
- Cleanup up expression and table formatting. Fixed case for TIME, temperature, Frequency.
- Still need format for tlines, consistent use of Courier for ML statements.

08/31/01 – Lynne Green

- Rev 0.3a
- Merged Sections 1 and 2.
- Removing things not specific to the macro language. (Moved this stuff to new Section for now.)

08/10/01 – Lynne Green

- Rev 0.2c
- Updated primitives list and formats, based on meeting of 7/31/01
- Created text in Section 6.
- Section 3: Started a list of reserved words.
- Section 9: Started a list of "Future Extensions".
- Examples and explanatory text still need to be checked for consistent Case use.

06/19/01 – Stephen Peters

- Rev 0.2b
- Add some basic formatting, sections 7 and 8 work
- Start on section 5, still very much work in progress

11/15/2000 -- Stephen Peters

- Initial Revision 0.1

---

---

Section 1

GENERAL INTRODUCTION

---

---

This section gives a general overview of the remainder of this document.

Section 2 contains general information about the IBIS Macro Language (IBIS-ML) and its relationship to IBIS. Section 3 presents the general syntax rules and guidelines for creating an IBIS-ML file.

The formal specification of IBIS-ML follows in sections 4 through 8. Section 4 presents a general overview of the language while section 5 describes the circuit elements and nodal netlist syntax. Section 6 concerns itself with expressions. IBIS-ML control keywords and circuit building control constructs are detailed in section 7. Finally, section 8 discusses data types and the rules for creating and using the data templates in the IBIS models.

---

---

 Section 2

---

---

 SYNTAX RULES AND GUIDELINES
 

---

---

This section contains general syntax rules and guidelines for IBIS-ML files as well. It also contains a list of reserved words.

## 2.1 Syntax Rules

1) Unless overridden by a rule stated below, IBIS-ML shall follow the general syntax rules and guidelines as stated in the latest EIA/IBIS-656 specification.

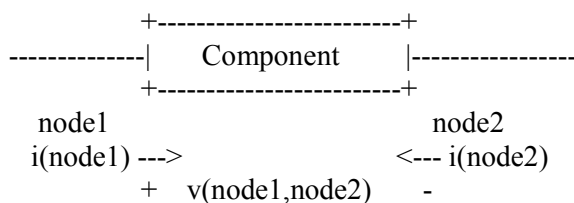
2) When creating an IBIS-ML library file, all lines that occur before the opening keyword or after the [End] keyword shall be treated as comments and ignored by the parser. The intent of this rule is to facilitate the inclusion of HTML tags, revision control headers, etc. at the beginning of the file.

3) Comments are not allowed between the opening and closing parentheses of any expression, whether that expression is on one or more lines of the IBIS-ML file.

4) Multiple definitions of any symbol are illegal. Specifically, multiple definitions (either redundant or alternate) of an object type of the same object class (within the same scope) are illegal.

5) The following rules govern reference directions for components and nodes:

- All currents are considered positive if they are into a component pin.
- Voltages can only be measured between pairs of nodes.
- The voltage between a pair of nodes is measured from the first node to the second node. The value will be positive if the first node is at a higher potential than the second node, and negative if the first node is at a lower potential.



6) All instance labels must begin with a letter, and shall consist only of alphanumeric characters and the underscore (`_`). Node names shall be either numbers or else an alphanumeric beginning with a letter, and underscores are permitted.

7) There are no global node names. The nodes POWER, GND, and 0 are local to the component where they are used. An EDA tool is required to convert these local node names if necessary to prevent conflict with the tool's global node names.

## 2.2 Reserved Words and Pre-defined Constants

The following words are reserved and shall not be redefined or used in any other context that that documented by this specifiaction. A parser error will be generated if any are redefined.

Language operation: ||, &&, foreach, if, else, end, while, assert, trigger, inherit, node, select, case, export

Circuit elements: resistor, capacitor, inductor, vsource, isource, vcontrolled, icontrolled, tline

Mathematical operators: Delay, sin, cos, tan, arcsin, arccos, arctan, exp, exp10, log, log10,

Simulator state variables: Time, Temperature, Frequency

The following words are reserved within the scopes of a [Define] keyword (a macro).. A parser error will be generated if any are redefined anywhere in an IBIS-ML file.

Case insensitive, show uppercase in the document.

Netlist Operators: OPEN, SHORT

Circuit element parameters: Z0, TD, LEN, R, L, G, C, V, I, Rlcgfile

The language supports a number of predefined constants that can be used in expressions. As with reserved words, these symbols should not be redefined. However, no parser warning is generated if any of these is redefined.

PI - ratio of a circle's circumference to its diameter

Q\_E - magnitude of an electron's charge

KB - Boltzman's constant

EPS\_0 - Dielectric constant of free space

Note: It is recommended that an EDA tool use sufficient precision accuracy for these constants, and ensure enough digits are used.

This is in addition to words already reserved in the IBIS specification.

---

---

Section 3

THE [DEFINE] KEYWORD

---

---

This section of the LRM describes the [Define] keyword and its use.

This requires more than just the syntax, since it provides so much of the flexibility for IBIS-ML modeling. Examples to be included.

Actually, the [Define] construct is covered in what is now section 7. What this section should be in the 'statement of intent', and is placed before the general syntax rules

This is what was agreed to on the chapter outline.

---

---

## Section 4

---

---

### GENERAL OVERVIEW

---

---

#### 4.1 Introduction

IBIS-ML is a macro language intended for use with IBIS, utilizing a nodal based syntax similar to that used for traditional circuit simulators. The user builds a nodal description with components. The components and their values (including tables and equations) represent (mimic) the behavior of a simulation object (such as an I/O buffer). The nodal description is not required to represent any actual circuit or structure.

A simulation object is described to the simulator in two sections: The [Define Model] section defines the behavioral structure of the model type (such as an I/O buffer). The [Model] section contains the data and tables for the model (such as a specific buffer model).

In the IBIS macro language, the “model type” is a behavioral description or template, with the data values being supplied by the IBIS data file that contains the data for one specific model of that model type. In other words, the [Define Model] creates a parameterized template, and the [Model] supplies the values for the parameters. For example, an I/O buffer has type inout, and the associated data file may contain data values for buffer1 and buffer2. The IBIS model template is in the IBIS-ML library file, and the data is in the IBIS data file (these may be the same or different files).

IBIS-ML also contains programming constructs (such as if-then-else, select-case) and other directive constructs, which are required for support of IBIS data structures and models. These constructs tell the EDA tool how to use the data in the model (that is, what data types and/or topological constructs to include in the model). These programming constructs allow the EDA tool to examine the data in the IBIS data file and then build an object’s netlist based on the values found.

#### 4.2: A Buffer Example

Following is a two-part example showing how the models for a family of receivers might be constructed. The purpose of this example is to introduce the reader to the syntax and structure of the language *as a whole* so that the reader may gain an overall understanding and ‘feel’ for the language before diving into the details presented in the later sections.

As described in this specification, an object definition begins with the [Define “class name”] keyword and is terminated with the [End “class name”] keyword. In this case, the object being defined is of the class ‘Model’, while the class type is ‘simple\_receiver’. The definition of ‘simple\_receiver’ includes the optional port list.

<is port list required to be explicit in the definition? how about the parameter list?>

Let’s assume the receiver family has conventional power and ground clamps plus an optional pullup resistor to the power rail. The circuit-model creator decides that the “circuit” (behavioral object) will include the clamps, the pullup resistor, and the input pad capacitance. The data file must then contain I/V curves for the clamps, the value of the pullup resistor, and the value of the pad capacitance.

The receiver function described above can be expressed using the IBIS-ML as follows:

```
[Define Model] simple_receiver (pwr, gnd, input)
capacitor io_cap (input GND) C=C_comp
resistor term_res (input POWER) R=(R_pullup || OPEN)
iconrolled pclamp (input POWER) (input power)
    I = [POWER Clamp] (v(input,power))
iconrolled gclamp (input GND) (input gnd)
    I = [GND Clamp] (v(input,gnd))
[End Model] simple_receiver
```

The corresponding data section (an IBIS [Model] section that contains the data for simple\_receiver) is contained in the IBIS data file, and is as follows:

```
[Model] xyz
model_type simple_receiver
C_comp 3pF 2pF 4pF
R_pullup 50 47 55
[Supply Voltage] 3.3v 3.0v 3.6v
[Power Clamp]
| volts I(typ) I(min) I(max)
-3.3v -100ma -80ma -120ma
-2.5v <etc>
[Gnd Clamp]
| volts I(typ) I(min) I(max)
-3.3v 100ma 80ma 120ma
-2.5v <etc>
```

To use this model for a buffer, the model xyz is instantiated (such as in an IBIS component pinlist or a schematic), and its model\_type (simple\_receiver) tells the simulator which model\_type (template) to use. This model\_type template also tells the EDA tool that there are three local nodes (pwr, gnd, input) that can be used for connecting to other components. Note that there are no global nodes in IBIS-ML.

The simple\_receiver is built from components. Each component has a type (capacitor), a name (io\_cap), nodes (input) and a value (C\_comp). Component values can be symbolic constants (C\_comp), fixed values (3.3v), or tables ([GND Clamp]).

The symbol ‘||’ is the logical OR operator. In the [Define Model] section, it tells the EDA tool to use the value of R\_pullup if it is provided, and to use the value OPEN otherwise. Since the value for the symbolic constant R\_pullup is defined in xyz, the buffer contains the pullup resistor in the circuit topology.

Now we are ready to look at each statement in the [Define Model] section. There are IBIS-ML statements, which are read as follows:

```
capacitor io_cap (input gnd) C=C_comp
```

This line instructs the EDA tool to include a capacitor with the name ‘io\_cap’ between the nodes ‘input’ and ‘gnd’ of the [Model] simple\_receiver. The capacitor value is assigned (C=) to the symbolic name (parameter) ‘C\_comp’, as specified by C= C\_comp. In this example, the parameter ‘C\_comp’ in the [Model] data section supplies the value for the symbolic name

'C\_comp' in the [Define] section. Symbolic names are placeholders for values (strings, numbers, arrays of values, etc.), where the specific values are given in the data section.

```
[Model] xyz
model_type simple_input
.
.
C_comp 3pF 2pF 4pF
.
.
[end Model]
```

The next line of IBIS-ML defines a resistor named term\_res, between the nodes input and power.

```
resistor term_res (input POWER) R=(R_pullup || OPEN)
```

The resistor value is assigned (R=) to the symbolic name (parameter) 'R\_pullup', as specified by R=(R\_pullup || OPEN). This is be read "R equals the value of the R\_pullup parameter or, if the R\_pullup parameter is not present, then R equals OPEN", where "OPEN" is an open circuit. This construct allows a user to specify an optional parameter.

The next line defines a controlled current source named pclamp.

```
icontrolled pclamp (input POWER) (input power)
I = [POWER Clamp] (v(input,power))
```

This controlled source uses the voltages at the nodes in the second set of parentheses ("input" and "power") as the controlling nodes. The term I= indicates that the source produces a current. The table name is [Power Clamp]. The table name matches the table name in the data file, including the brackets. The table name is followed by the table input lookup values, enclosed in parentheses.

The next line describes a second controlled current source. This is the other clamp, and it uses a different control node pair and a different table from the first clamp.

```
icontrolled gclamp (pin GND) (pin gnd)
I = [GND Clamp] (v(pin,GND))
```

The next keyword that follows the [Model] data begins the next section, and ends the model.

### 4.3 IBIS-ML Library File

A combined behavioral and data model description, such as the example above, may be contained in a single IBIS file describing a component, package, model definition, and model data. Alternatively, a user may collect one or more IBIS-ML model descriptions, and put them in a separate file. These model definition files have an .ibs extension.

Any file containing only IBIS-ML model definitions is referred to as an IBIS-ML *library file*. A library file can be included in another IBIS file by use of the [Include Library] keyword.

As with any IBIS file, an IBIS-ML library file must begin with a legal header section (as defined by the IBIS specification) and is terminated with the [End] keyword. Between the end of the header section and the [End] keyword are the [Define]/[End Define] keywords for each object defined as well as the IBIS-ML statements themselves.

For reference, a tree diagram outlining the contents of a IBIS-ML library file is presented here. Both the required and optional keywords of the header sections are shown. The global keywords [Comment Char] and [Include] are not shown.

```

can group [Define] statements in a library file,
or put [Define] inline with data.
Just like in pkg or ebd files.
/-- Start of File
/ comment text
| Header section
| /-- [Begin Header] or [IBIS- Ver]
| |-- Header statements
| \-- [End Header] or next keyword
|
| /--[Define "class name"]
| |-- IBIS-ML statements
| \--[End "class name"]
\-- [End]

```

When an IBIS Data File is a separate file, it must have its own header and [END] keyword, and one or more model data sections.

```

/-- Start of File
/ comment text
| Header section
| /-- [Begin Header] or [IBIS- Ver]
| |-- Header statements
| \-- [End Header] or next keyword
|
| /--[Model]
| |-- Model data section
| \-
| /--[Model]
| |-- Model data section
| \-
etc.
\-- [End]

```

If both the Library and Model Data are in the same file, then only one header and one [END] are used.

---



---

 Section 5

---



---

 BASIC CIRCUIT ELEMENTS
 

---



---

This section of the LRM describes the basic circuit components (primitives) that can be used in a [Define Model] template to create a model topology (behavioral circuit description). This section also describes the nodal syntax used for interconnecting circuit components into behavioral models

The primitive components include the traditional resistance, inductance, capacitance, independent and controlled source components, and transmission lines. By allowing these components to be described using tables and equations as well as traditional values, this set of components can support all IBIS constructs.

### 5.1 Basic Circuit Element

Table 1 below lists the basic circuit components, along with a brief description of each. For each component, the value may be an expression or table involving constants, symbolic values, node voltages, TIME, temperature, and/or Frequency. A table may have one or more dimensions. An expression is a mathematically valid expression, as described in Section 5.

**Table 1 – Basic Circuit Components**

Element Name	Description
resistor	R = <value> (constant, function, or table)
capacitor	C = <value> (constant, function, or table)
inductor	L = <value> (constant, function, or table)
isource (current source)	I = <value> (constant, function, or table)
vsource (voltage source)	V = <value> (constant, function, or table)
Vcontrolled	V = <value> (function or table)
icontrolled	I = <value> (function or table)
tline	a) len=<value>, Z0=<value>, TD=<value>, all three required b) len=<value>, R=<value>, L=<value>, C=<value>, G=<value>, L and C required, R and G optional c) len=<value>, RLCGmatrix=<matrix name or file name>
One or more tlines	len=<value>, RLCGfile=<table name>
Subcircuit	A subcircuit with explicit ports and explicit parameters

Table 2 lists some event-detection constructs that are required to support existing IBIS model constructs.

**Table 2 – Circuit Components for Backwards Compatibility**

Name	Description
Trigger (Boolean condition)	Event output
Alarm (Boolean condition)	Event output
Driver	?

Reshape <?>	generates a digital pulse, reshaped from its input ?
Delay (in, out) TD	out = in, delayed by TD
Voltage controlled delay	Out = in, delay is a function of V (future release) ?

## 5.2 Basic Components

This section describes each of the basic components. As in IBIS, the SI system of units is used (volts/amps/ohms/seconds) for all components.

### resistor

A resistor is a two-terminal circuit component that represents the ratio of voltage across its terminals to the current through the component ( $R=V/I$ ). Units are Ohms.

### capacitor

A capacitor is a two terminal circuit component that accumulates charge, where the value of capacitance is a measure of the amount of charge accumulated for a given rate of change of voltage across its terminals ( $I = C*d\_dt(V)$ ). Units are Farads.

### inductor

An inductor component is a two terminal component that accumulates magnetic flux, where the value of inductance is a measure of the amount of magnetic flux per a given rate of change of current through the component ( $V = L*d\_dt(I)$ ). Units are Henries.

### vsource

A vsource is a two terminal circuit component representing an independent voltage source. The voltage (potential difference) between the terminals of vsource is referenced from the first pin to the second pin. Units are Volts.

```
Pin1 >--<vsource>--> Pin2
+      Voltage      -
```

### isource

An isource is a two terminal circuit component representing an independent current source. The positive current is into the component's pin 1. Units are Amps.

```
Pin1 >--<isource>--> Pin2
Current ---->
```

### vcontrolled

A vcontrolled is a controlled voltage source. In IBIS-ML Version 1, the output voltage is a function of the voltage between the controlling nodes, and current control is not permitted. The component allows any number of controlling nodes. Units are Volts.

```
vcontrolled = function(controlling node voltages)
Pin1 >--<vcontrolled>--> Pin2      voltage_in(Pin3, Pin4,...)
+      voltage_out      -
```

**icontrolled**

An icontrolled is a controlled current source. In IBIS-ML Version 1, the output current is a function of the voltage between the controlling nodes, and current control is not permitted. The component allows any number of controlling nodes. Units are Amps.

```
Pin1 >--<icontrolled>--> Pin2      voltage_in(Pin3, Pin4,...)
Current_out ---->
```

**tline**

A tline is an N-terminal component representing one or more transmission lines. Each transmission line includes length (meters), impedance (ohms), and delay (seconds), and may optionally include loss and frequency dependence. The individual transmission lines may be coupled or uncoupled. Section 5.3 describes the allowed ways to express tline characteristics.

A single tline can be described using the delay or the RLCG values at a single frequency. Note: the length and the per unit length must use the same units (such as meters, inches, or mils).

Z0 and ZO shall be accepted.

tline arguments shall be separated by and comma and/or one or more spaces.

Note that all TD, R, L, C, and G values are per unit length.

To represent more than one transmission line (coupled lines), or for modeling frequency dependent loss, the RLCG matrix format is required. This used the same RLCG matrix format as defined in the IBIS Connector specification.

```
driver
?
```

### 5.3 Instantiating Basic Circuit Components

The syntax for using the basic circuit components in a [Define Model] scope (netlist) is shown below:

```
component_name instance_label (node list) symbol=value
```

```
controlled_component_name instance_label (node list) (control node list) symbol=value
```

A space is required between the component name and the instance label. A comma or space is required between the nodes in a node list. Parentheses are required around node lists and around argument lists. A comma or space is required between the arguments for a table.

Additional spaces may be used. For example, additional spaces are used in many of the examples in this document, to make them more readable.

Comment characters may not appear inside the component definition, even if the component statement takes more than one line. A comment may follow at the end of the last line of the component statement.

**Examples of valid component statements:**

| the minimum number of required spaces?

```
resistor this1(node1,2)R=10k
```

| alphanumeric node names, comma separated node list

```
resistor diode1 (in,out) I=(1e-12*exp(v(in,out)/0.26))
```

```

|numeric node names, space separated node list
capacitor that2 (1 2) C=2e-12
| table arguments must be separated by comma or space
isource diode2 (source gnd) i = [diode_table] (v(source,gnd), temperature)
| tline with space separation of arguments and optional additional spaces
tline tl1 (1 2 3 4) len = 2 Z0 = 50 TD = 2n
| tline with comma separation of arguments and no optional spaces
tline tl2(1 2 3 4)len=2,R=0.1,L=2n,C=25p

```

**Examples of invalid component statements:**

```

resistor dummy1 (node12) R=10k | requires two nodes
isource dummy2 (in,out) |
comment in the middle of the two-line statement
I=(1e-12*exp(v(in,out)/0.26))

```

The allowed *component names* are listed in Tables 1 and 2 above. Component names for new components and subcircuits created using a [Define] statement may also be used. The component name is followed by the required blank space.

Following the component name is the *instance label*, which uniquely identifies a particular instance of a component within its scope (such as within a subcircuit or module). The instance label is optionally followed by a blank space.

Following the instance label is a whitespace-separated *node list*, which specifies the external nodes or terminals of the component. The node list is required to be surrounded by parentheses. The nodes in the list shall be separated either a comma or one or more blank spaces. The node list is position sensitive (that is, the first node in the list of an instance is matched to the first node in the component primitive, the second is matched to the second, and so forth). The node list may be followed by one or more spaces.

Finally, the component is assigned a value. This assignment is shown as <symbol>=<value>, where symbol is one of the following:

Value is a:	Symbol to use
Resistance	R, V, or I
Capacitance	C
Inductance	L
Voltage	V
Current	I

Note that the equals sign between the symbol and the value is required. One or more blank spaces may be used on either or both sides of the '=' character.. The '=' is followed by a value, which may be scalar, a symbolic constant, a table name with an argument list, or an expression. When used, an expression must be enclosed in parentheses.

Following are some examples of component instantiation:

```

| simple value
resistor example1 (node_a node_b) R=1k
| symbolic constant
capacitor example2 (alpha baker) C=C_comp
| equation with a symbolic value
inductor example3 (1, 2) L=(L_pkg + 1.5n)
| table reference
isource (out_pos out_neg) I = [pullup] (v(out_pos,out_neg))

```

```

| equation with node voltages
icontrolled (out1 out2) (in1 in2 in3)
            I = (1e-3*V(in1,in2) + 2e-3*V(in2,in3)/VCC)

tline tlineA (left1 0 right1 0) len= Z0=50 TD=5n
tline coupled_pair (left1 left2 left3 left 4
right1 right2 right 3 right4) RLGCFfile = <matrix name>
tline tlineC (1 2 3 4)S_matrix=<s-param matrix name>

```

## 5.4 Assigning Values To A component

### Values

All component values can be expressed as constants, symbolic values, expressions, or tables. An expression can depend on state variables (TIME, temperature, Frequency), symbolic constants, the component's node voltages, and parameters. Currents and external voltages cannot be used in expressions.

An IBIS-ML expression may use the following operations: arithmetic (+ - \*/), trig (sin, cos, tan, arcsin, arccos, arctan), exponential (exp, exp10, log, log10). When used, an expression shall be enclosed in parentheses.

The value assigned to a component can be expressed in a number of ways. The value may be an explicit value (a constant), a symbolic value, a value derived from a table lookup, or an expression made up of any or all of the above. Some examples are shown below:

```

| Assigning an explicit value
resistor r1 (pad gnd) R=10K
| Assigning a scalar symbolic value
capacitor c_output (output gnd) C=C_comp
| Assigning a built-in symbolic value in an expression
resistor r1 (pad gnd) R= (10K*(temperature-273))
| Assigning a value from a table lookup
icontrolled clamp (output POWER) (output POWER)
            I=[Pullup] (V(output,POWER))
| Assigning a value from a two dimensional table lookup
vcontrolled ramp (out1, gnd) (vin1)
            V=[Ramp_data] (V(in1,gnd),TIME)

```

### Controlled Source Components

A controlled source has a list of controlling nodes as part of its format. The output voltage value is calculated as follows:

If there are exactly two controlling nodes and the <value> is a constant, then the output is constant\*V(in1,in2). the constant may be either a numeric value or a symbolic constant

Examples:

```

icontrolled is1 (1, 2) (3, 4) I=(0.3*V(3,4))
vcontrolled vs1 (5, 6) (7, 8) V=(0.3*V(7,8))

```

In all other cases, the value is either an expression or a table. The voltages on the control nodes may be used individually (such as V(3)), or a voltage difference between two nodes may be used (such as V(3,4)). The format for a voltage function is:

V(node1, node2) | voltage at node1 relative to node2

Examples:

```
vcontrolled vc1 (vref gnd) (vcc) V=(V(vcc,gnd)/2)
icontrolled clamp (output vcc) (output,vcc)
          I= [Pullup] (V(output,vcc))
```

## Tlines

For a single, uncoupled transmission line, the transmission line characteristics can be defined using len (length), Z0 (characteristic impedance), and TD (delay per unit length). A single transmission line may also be defined using RLGC or s-parameter matrices.

For a set of coupled transmission lines, the characteristics must be defined using either RLGC or s-parameter matrices.

## 5.4 Instantiating a Subcircuit

The syntax for instantiating a subcircuit into a model (netlist) is similar to instantiating basic components. The sub-parameter “subcircuit” is used to indicate that the string subckt\_name contains the name of the behavioral model to use.

```
Subcircuit      subckt_name      instance_name      (node      list)
<(parameter list)>
```

The component name ‘Subcircuit’ indicates this component is defined as a model, which is named subckt\_name. These are followed by the instance name and the node list. This is followed by an optional *parameter list*, which is enclosed in parenthesis.

The node list is positional. the first node in the node list is connected to the first node in the behavioral mode, the second to the second, and so forth.

The parameter list mechanism allows the user to customize the subcircuit on an instance by instance basis. Since symbolic values are limited to their local model in scope, except for the pre-defined state variables, all parameters must be explicitly passed when they are to be used in a subcircuit.

---

---

 Section 6

---

---

 EXPRESSIONS, FUNCTIONS, TABLES, AND MATRICES
 

---

---

## 6.1 Symbolic Values

An symbolic value is a function that evaluates to a *constant* numeric value that is known at the beginning of simulation. A symbolic value (parameter) must be defined before it is used (for example, in determining a component value). A symbolic value cannot depend on TIME, or anything that can change with time. A symbolic value cannot depend on any node voltages or pin currents.

The format for an symbolic value is *define* <label> = <value>. “define” is the sub-parameter that tells the system a constant is being defined. “<label>” is the name for this symbolic value; names must be unique within their scope. The “=” sign is required before <value>.

The value can be a single number, a row of numbers, an expression, or a table lookup. Single numbers follow the same rules as for IBIS 3.2 values, which permits use of engineering and scientific notation. A row of numbers follows the IBIS 3.2 syntax for sub-parameters, with columns in the order in `typ/min/max/<others?>`. An symbolic value is evaluated mathematically once per simulation. A table lookup is performed once per simulation.

Examples of symbolic value use:

```
| a simple value
define Omega = 3
| a row of values
define Param2 = 5.5 5.0 6.0
C_comp 2p 1p 3p
| An symbolic value
define DegreesKelvin = (temperature + 273)
| table lookup
define R_pullup = [R_vs_T], temperature
```

Symbolic values can be used in expressions, as long as the symbolic value is defined before it is used.

## 6.2 Expressions

An expression is a mathematical expression that is evaluates to a single value. An expression can depend on TIME. An expression should be evaluated at each time step.

An expression is always preceded by an “=” sign and must be surrounded by parentheses.

An expression may take more than one line. The end of the expression is when the total number of left parentheses and right parentheses are equal. If a keyword is encountered before the last right parenthesis, the parser must report an error at the keyword line.

Examples of equations used in determining component values:

```

| simple expression
inductor example3 (1, 2) L=(2.7n + 1.5e-9)
| expression used in argument of a table reference
isource (out_pos out_neg) I = [pullup] (V(outpos,outneg))
| expression using symbolic value and controlling node
voltages
icontrolled (out1 out2) (in1 in2 in3)
I = (1e-3*V(in1,in2) + 2e-3*(V(in2,in3)/VCC)

```

## 6.3 Tables

A table is two or more blocks of data, with each data block being formatted with one column for the first independent variable (such as buffer pad voltage), and three columns for the output values (typ/min/max) (such as clamp current). A table can depend on more than one input, in which case the table has one block of data for each additional input value set.

Examples include the IV and VT tables defined in IBIS 3.2.

## 6.4 Matrices

Two types of matrices are supported. RLGC matrices are used for a lumped-component model, while s-parameter matrices can be used for both lumped and distributed models. S-parameter data can also be obtained from measurement.

RLGC matrices, as defined in the IBIS Connector Specification, may be used for any “lumped” characterization. In particular, RLCG matrices can be used for tline and connector components.

S-parameter matrices are used to represent frequency-dependent model data. An s-parameter matrix can be used for any passive structure, including lumped circuits and distributed components (such as tlines). The matrices must be supplied in Touchstone format, which is available in several field solvers as well as from ATE vector-network analyzers. If the Touchstone file is a separate file, it should be formatted as an IBIS-ML library file, and then included using the [Include Library] keyword.

Coupled inductors can be represented using either RLGC matrix or s-parameter matrix format.

## 6.5 Functions

A function is an expression that is not contained on the same statement as the component. A function is called by reference. The call to the function shall have the same number of arguments as on the function definition.

Functions will (not) be supported in the first release of the IBIS Macro Language.

---

---

Section 7

THE DEFINE CONSTRUCT

---

---

This section of the LRM describes the “Define” control/programming construct and its use.

This requires more than just the syntax, since it provides so much of the flexibility for IBIS-ML modeling.

Need to make clear how this differs from the [Define] keyword.

Section 8

TEST AND CONTROL CONSTRUCTS

This section of the LRM describes the test and control constructs available in IBIS-ML. These constructs allow the user to control how the circuit description is processed on read-in. For example, the user can tell the EDA tool to check for the existence of specific keywords or sub-parameters in an object’s IBIS data set, and/or make decisions based on the values of variables. It is important to emphasize that these test and control statements are evaluated at model read-in or compile time. In that sense these control statements are like the preprocessor directives for a standard programming language such as C.

The following test and control statements are available:

Name	Short Description
assert	display a message based on the results of an expression
define	assign a value to a variable
foreach	create an array of statements then fill them in with data
if, else if, else, end	test an expression
if	
inherit	include a previously defined object into the one being defined
node	create an internal node
select, case	select a data set or object configuration based on user input
export	

The syntax and use of the above statements are defined in sections 7.1 through 7.8 below.

## 8.1 The “assert” Statement

---

### Abstract

The "assert" statement evaluates an expression then displays a message if the expression is false.

---

### Syntax

```
assert ( <expression> ) “message”
```

---

### BNF

```
assert: 'assert' '(' expression ')' "" message ""
```

---

### Description

The assert statement is used to perform checks on the data supplied in the data section of an IBIS data file. The condition is tested at compile time and an error is issued, with the indicated message, if the condition tested evaluates to false. The message part of the assert construct is a text string surrounded by double quotes, and is required. The simulator is not required to continue after a failure, because the data is considered to be defective.

It must be possible to fully evaluate the expression at compile time and the expression must evaluate to a Boolean true or false value. For a check that is evaluated at run time, see "alarm".

---

### Example 1

In this example assert is used to limit the range of numeric values allowed for the symbolic name ‘Rload’.

```
[Define Model] load_resistor (pin gnd)
resistor Rload (pin gnd) R=Rload
assert (Rload >= 10) "Rload too small"
assert (Rload <= 100) "Rload too high"
[End Model] load_resistor
```

### Example 2

In this example, the string variable ‘Enable’ may have one of two values: “Active-Low” or “Active-High”. All other values are illegal. The assert statement in the ‘else’ part of the if-else construct tests ‘Enable’ for the value “Active-High” and tells the EDA tool to display the message “Enable: illegal value” if the expression (Enable == “Active-High”) is false.

```
[Define Model] xyz
.
.
if (Enable == "Active-Low")
subckt invert (en 0 enable_pin) inverter
```

```

else
  assert (Enable == "Active-High") "Enable: illegal value"
  subckt buffer (en enable_pin) non-invert
end if
.
.
[End Model] xyz

```

## 8.2 The “define” Statement

-----  
 Abstract

The define statement assigns a value to a symbolic name.

-----  
 Syntax

define name = <expression>

-----  
 BNF

define : 'define' name '=' expression

-----  
 Description

The define statement assigns a value to a symbol. The assigned value may be a constant or the results of evaluating an expression at compile time. The expression must be fully evaluated at compile time and the define statement must occur before the name being defined is used.

Defines may be used in an expression anywhere a scalar variable or scalar constant may be used.

-----  
 Example 1

In this example the user assigns a constant value to the symbol “default\_z”.

```
define default_z = 50
```

Example 2

Here, the symbol VT is assigned the results of an expression evaluation.

```
define VT = .8617087e-4 * (temperature+ 273.15) || .026
< need more significant digits than 2?>
```

### 8.3 The "foreach" Statement

---

#### Abstract

The "foreach" statement operates across an array of statements. These statements contain variables that are filled in with data from a corresponding 'foreach' data array in the IBIS data file.

---

#### Syntax

```
foreach <index> in <pointer to data structure in IBIS data file>
  IBIS-ML statement
.
.
  IBIS-ML statement
end foreach
```

or –

```
foreach <index> in <pointer to data structure in IBIS data file>
  IBIS-ML statement
.
.
  IBIS-ML statement
else
  IBIS-ML statement
.
.
  IBIS-ML statement
end foreach
```

---

#### BNF

```
false_part : 'else' '\n'
            statement

foreach : 'foreach' index 'in' key_name '\n'
         statement
         false_part
         'end foreach' '\n'
```

---

#### Description

The foreach construct iterates across an array of data, allowing the user to place items from the array into statements. The foreach statement is constructed as follows:

A line beginning with "foreach", followed by a symbolic name used as an index, followed by the word "in", followed by a *key\_name* (in the form of an IBIS keyword) that points to an array of data in an IBIS data file. Note that the array of data is assumed to be formatted as shown in the 'foreach' data array description in Section 8.

Any number of IBIS-ML statements. These statements include *substitution symbols*. Substitution symbols are placeholders for the data from the IBIS data file or for the foreach index variable itself.

An optional "else" section, identical to the "else" section of an "if" statement.

The line "end foreach" terminate a "foreach" construct.

The "else" section introduces a section that is used only when the key\_name is not present in that object's data section. Note that in the case, the IBIS-ML statements following the 'else' clause may not contain the substitution symbols (e.g. there is no data to substitute).

Substitution symbols are of the general form \$<symbol>\$ where <symbol> is either the index variable of the foreach statement or an integer that represents the fields in an array of data. The symbol \$0\$ represents the first field in a line of data , \$1\$ represents the next field, and so on. The substitution symbols may be used in the middle of a string. Example: Foo\$1\$Bar.

-----  
Uses in IBIS 3.2

"Foreach" arrays are used without an else section to define the array of submodels for the "[Add SubModel]" construction.

They are used with an else section for "[Driver Schedule]". The else section is used only when there is no "[Driver Schedule]" keyword.

-----  
Example 1

In this example, a model is created that consists of either an array of parallel resistors and subcircuits, or a single capacitor 'C\_shunt'.

```
[Define Model] parallel (pin gnd)
foreach iii in [A_Batch_Of_Data]
  resistor R$iii$x (pin a$iii$) R = $1$
  subckt X$iii$x (a$iii$ gnd) $0$
else
  capacitor Cx (pin gnd) C=C_shunt
end foreach
[End Model] parallel
```

The corresponding [Model] data set is:

```
[Model] xyz
Model_type parallel

[A_Batch_Of_Data]
Model_A 120 slow
Model_B 176 fast
[End A_Batch_Of_Data]
C_shunt 100p
```

[End Model] xyz

Applying the foreach construct to the above data set yields:

```
resistor R1x (pin a1) R = 120
subckt X1x (a1 gnd) Model_A
resistor R2x (pin a2) R = 176
subckt X2x (a2 gnd) Model_B
```

Note that because the array [A\_Batch\_Of\_Data] exists, the 'else' part of the foreach statement was not executed. Therefore, the netlist does not include C\_shunt.

#### Example 2

Alternately, the above [Model] data set could be as shown below:

```
[Model] abc (pin gnd)
Model_type parallel
C_shunt 100p
[End Model]
```

This results in an equivalent circuit of:

```
capacitor Cx (pin gnd) 100p
```

Because the [Model] did not include the keyword [A\_Batch\_of\_Data], there is no resistor or subckt.

## 8.4 The "if" Statement

-----  
Abstract

The "if" statement, and its companions "else if", "else" and "end if" are used to choose sections of a macro based on the results of a test.

-----  
Syntax

```
if (<expression>)
  IBIS-ML statements
end if
```

or –

```
if (<expression>)
  IBIS-ML statements
else
  IBIS-ML statements
```

```

end if

or –

if (<expression>)
  IBIS-ML statements
else if (<expression>)
  IBIS-ML statements
[else if (<expression>)
  IBIS-ML statements
else
  IBIS-ML statements]
end if

```

-----  
BNF

```

false_part : 'else' '\n' net_list
            | 'else if' '(' expression ')' '\n'
              net_list
              false_part
            | /* nothing */

```

```

if : 'if' '(' expression ')' '\n'
     net_list
     false_part
     'end if' '\n'

```

-----  
Description

The "if" statement is constructed as follows:

1. A line beginning with "if", followed by an expression in parentheses that evaluates to a Boolean true or false at compile time.
2. A "true" section, consisting any number of statements.
3. Zero or more "else if" sections consisting of:
  - 3a. A line consisting of "else if" and an expression, like statement 1 above.
  - 3b. A set of any number of statements, like statement 2 above.
4. An optional "else" section consisting of:
  - 4a. The line "else".
  - 4b. A set of any number of statements, like statement 2 above.
5. The line "end if" to terminate it.

The selection is done at compile time, along with expression evaluation. Its behavior is similar to the "#if" preprocessor directive in C.

The "select" statement sets up a named scope; the "if" statement does not.

-----  
Uses in IBIS 3.2

This is used wherever decisions are made based on parameters supplied.

1. The phase flip relating to the "Enable" and "Polarity" keys.
2. Determining whether power is supplied internally or externally.
3. Determining whether submodels are applied "driving" or "non-driving".

-----  
Example

## 8.5 The "inherit" Statement

-----  
Abstract

The "inherit" statement includes another object into the object currently being defined.

-----  
Syntax

inherit <base type>

-----  
BNF

inherit : 'inherit' literal

-----  
Description

The "inherit" statement brings in another object as a base. The actual mechanism is a direct text substitution of the statements. If this inherit statement is the first statement in a [Define] block, and this [Define] block has no port list, the port list is also inherited.

-----  
Example

First, define a base object.

```
[Define Base] base_resistor (pin gnd)
resistor Rload (pin gnd) R=1k
[End Base] base_resistor
```

Now, use the inherit to include this object into 'driver'.

```

[Define Source] driver
.
.
inherit [Base]base_resistor
vsource Vout (pin gnd) V = [Rise](T-Trise) || [Fall](T-Tfall)
.
.
[End Source] driver

```

This is equivalent to:

```

[Define Source] driver (pin gnd)
.
.
resistor Rload (pin gnd) 1k
vsource Vout (pin gnd) V = [Rise](T-Trise) || [Fall](T-Tfall)
.
.
[End Source] driver

```

## 8.6 The "node" Statement

### Abstract

The "node" statement declares the existence of a new node.

### Syntax

```
node <node list>
```

### BNF

```
node_list : node_list node
          | /* nothing */
```

```
node : 'node' node_list '\n'
```

### Description

The node statement declares new (local) nodes that are not declared elsewhere. It is illegal to declare a node more than once in the same scope. Used 'node' to declare local nodes that are not seen outside, and a node statement is required if a node is not listed in a pin list. The scope of the node is the object being defined.

### Example

```
[Define Model] res_cap (pin1 pin2)
node internal
capacitor (pin1 internal) 120p
resistor (internal pin2) 1k
[End Model] res_cap
```

In this case, "internal" is a local node.

## 8.7 The "select" Control Statement

---

### Abstract

The "select" statement is used to choose between sections of a circuit based on a user specified attribute.

---

### Syntax

```
select (<attribute name>)
  default = <value>
  case (<value>)
    IBIS-ML statement
    .
    .
    IBIS-ML statement
  end case
  .
  .
  [case (<value>)
    IBIS-ML statement
    .
    .
    IBIS-ML statement
  end case]
end select
```

---

### BNF

```
default : 'default' '=' literal '\n'
        | /* nothing */

case :   'case' literal '\n'
        net_list
        'end' 'case' '\n'

case_list : case_list case
```

```

|/* nothing */

select : 'select' '(' env_var ')' '\n'
      default
      case_list
      'end select' '\n'

```

---

#### Description

The select statement is used to choose a specific internal configuration or behavior of an object, where the choice is based on the value of a user supplied attribute. This attribute usually represents some external condition or assumption that affects the way an object (or group of objects) behaves. For example, a SPST switch can be either open or closed, and a model for this switch would include data sets that describe the behavior of the switch in both open and closed cases. However, the EDA tool needs to know if the switch is to be modeled in the open or closed position (i.e. which description or data set to use). The select construct is the mechanism by which the EDA tool processing the model selects which data set to use, based on user input.

A “select” statement is constructed as follows:

A line beginning with "select", followed by the name of a user defined attribute, in parentheses. The attribute name must be in the form of a string variable.

An optional "default" line specifying which choice to make when the attribute is not defined or passed in. The default value must be the name one of the data sets associated with that attribute.

At least one "case" section consisting of:

A line beginning with "case", followed by the name, in parenthesis, of one the data set (i.e. a keyword in the IBIS data file) associated with that attribute. The value the case clause is testing also identifies the beginning of a scope, which is connected only for this "case".

A list of circuit components. Control lines are not allowed, except for "assert" and "inherit". Others are considered to be errors.

A line "end case".

The "end select" statement terminates the select construct.

It is important to note that the attribute is NOT part of the data set supplied by the IBIS data set. Rather, the user, usually through the EDA tool interface, supplies the attribute. The switching is done as part of preprocessing, in the same pass that selects data sets.

The value of the attribute passed in must match one of the listed case values. If there is a "default" statement, it is legal to not pass in this attribute. It is an error to specify a value that is not listed.

Note that the "select" statement sets up a named scope; the "if" statement does not.

---

#### Uses in IBIS 3.2

This is used to implement the "series switch" model type.

-----  
Example

In the example below a model 'on\_or\_off' is defined. The model has two signal pins and a control attribute "ctrl". A portion of the circuit is selected by the value of the attribute "ctrl".

```
[Define Model] on_or_off (pin1 pin2)

node t1
resistor Rser (pin1 t1) R = (R_series || short)
select (ctrl)
  case "[Off]"
    capacitor C1 (t1 pin2) C = C_shunt
  case "[On]"
    resistor R1 (t1 pin2) R = R_shunt
    capacitor C1 (t1 pin2) C = C_shunt
end select

[End Model] on_or_off
```

The [Model] data section must have an [Off] section and an [On] section. A scalar "R\_series" is optional outside of all [Off] and [On] blocks. The [Off] block must define a value for "C\_shunt". The [On] block must define a value for "R\_shunt" and "C\_shunt".

The two "C\_shunt" keys are in different scopes, so the fact that the names are the same is irrelevant.

The [Model] data section in the IBIS data file might look like:

```
[Model] switched_load
Model_type on_or_off

R_series 10

[On]
R_shunt 100
C_shunt 50p

[Off]
C_shunt 100p

[End Model]
```

This expands differently depending on the value of the argument "ctrl". If "ctrl" has a value of "[On]":

```
subckt switched_load pin1 pin2
Rser (pin1 t1) 10
```

```
R1 (t1 pin2) 100  
C1 (t1 pin2) 50p  
ends
```

If "ctrl" has a value of "[Off]":

```
subckt switched_load pin1 pin2  
Rser (pin1 t1) 10  
C1 (t1 pin2) 100p  
ends
```

---

---

 Section 9

---

---

 DATA TYPES
 

---

---

This section of the specification defines the data types available for symbolic names as well as the usage syntax within an IBIS model data section or object description. In brief, symbolic names are grouped into one of the four allowable data types: string variables, scalar numeric variables, single tables (single index tables), and table groups (multiple index). In addition, the following organization structures are available: scopes and "foreach" arrays.

The following sections provide details on these data types and their usage.

## 9.1 Strings

---

### Abstract

This section describes variables with string values

---

### Description

A string variable is a variable whose value consists of a string of alphanumeric characters. The value must begin with an alpha character. A string variable is used primarily in conditional statements in the macro language. Usually, there are only a few legal values.

---

#### Assigning values to string variables (data section usage)

A string variable and its value must be expressed on one line, with the variable and its value separated by at least one white space. An equals (=) character between the variable name and its value is optional. For example,

```
Position = OFF
Polarity non-inverting
```

are both legal ways of assigning a value to a string variable.

A string variable that is not defined is considered to be equal to the empty string.

---

#### Using string variables in a [Define] section

String variables are most often used in conditionals and asserts. When comparing or testing the value of a string variable the comparison string is surrounded by double quotes (“”) as shown below.

Example:  
if (Position == "OFF")

```

.
.
else
  assert (Position != "ON") "Illegal value for Position"
.
.
end if

```

The if statement compares the value of the string variable 'Position' with the string "OFF". Note the use of the double quotes. Likewise, the assert statement compares the value of Position with the string "ON".

## 9.2 Scalars

-----  
Abstract

This section describes variables with scalar numeric values.

-----  
Description

A scalar numeric variable is a variable whose value is a number.

-----  
Assigning values to scalar numeric variables (data section usage)

A scalar numeric variable and its value(s) must be expressed on one line, with the variable and its value(s) separated by at least one white space. An equals (=) character between the variable name and its value is optional. For example,

```

[Temperature Range] 25C 0C 100C
R_pullup = 50

```

are both legal ways of assigning a value to a scalar numeric variable.

Note that while a scalar numeric variable can have only one value at any one TIME, the user may specify several alternate values to assign to the variable. As shown for the [Temperature Range] variable in the above example, this is done by listing these alternate values in a row on the same line as the variable – i.e. the user constructs an array of data with multiple 'columns' but only one row. Column 0 is the variable name, column 1 is the first variable in the row, column 2 is the next variable in the row, and so on.

***Note: The mechanism for selecting which value to use at run time is TBD***

-----  
Referencing in a [Define] section

The use of the name of a scalar in an expression is interpreted as to substitute the value. As an example:

```
[Define Circuit] heater (b1 b2)
resistor R123 (b1 b2) R=[Load]
capacitor C123 (b1 b2) C=C_comp
[End Circuit] heater
```

```
[Circuit] xyz
Circuit_type heater
[Load] 50
C_comp 100u
[End Circuit]
```

In this example, the value of R123 is 50 ohms. C123 is 100 microfarads.

### 9.3 Tables

-----  
Abstract

This section describes numeric key-value tables.

-----  
Description

A numeric key-value table is a two dimensional array of data consisting of at least two rows and two columns. For each row of data, the independent variable is placed in the left most column (column 0) while the dependent variable(s) are placed in columns 1, 2 etc. The data in column 0 is the *index* for the data in that row. Tables are generally used to define nonlinear transfer functions and points on a waveform.

-----  
Constructing tables (data section usage)

Tables begin with a header identifying the table. This header is in the form of an IBIS keyword; i.e. [`<table name>`]. The square brackets surrounding the name are required.

On the line below the header the user may optionally specify the table's *behavior attributes*. There are three behavior attributes; Order, Below and Above.

The construct Order = `<integer>` specifies how the EDA tool should interpolate between points in a data column. Legal values for Order are 1, 2 or 3. 1 means use linear interpolation, 2 means use quadratic spline interpolation, while 3 means use cubic spline. If Order is omitted the EDA tool is free to choose the interpolation algorithm.

The construct Below = `<integer>` specifies how the EDA tool should extrapolate new data points that lie beyond the most negative value in the data column. Legal values for Below are 0, 1 and 2 and the value must be less than or equal to Order. A value of 0 means extend horizontally at the boundary value, 1 means to extend the slope as a straight line, and 2 means to extend the slope and second derivative. If Below is omitted, the simulator is free to choose the interpolation algorithm.

The construct Above = <integer> specifies how the EDA tool should extrapolate beyond the most positive value in the data column. Values and rules are the same as for the behavior attribute Below.

Following the behavior attributes the user may optionally specify additional <Name>=  
<value> pairs. These additional attributes can be used to group tables into the equivalent of multi-dimensional tables. The attributes are interpreted either as an additional index for multidimensional tables, or as data in a scope.

Following the optional behavior and other attributes, the user may optionally place column headings for the following table of numbers. If used, column headings are of the form <name> where name is an alphanumeric string. As with the data itself, each column heading must be separated by white space.

Following the attributes and headers is the data itself. This is a list of numbers, organized by row and column. If column headings are supplied, the number of columns must match the number of headings. The number of columns must be consistent within a table. The value "NA" indicates that a data point is omitted, but any column must have at least 2 non-NA values.

A table is terminated by the occurrence of the next keyword – i.e. a line beginning with '[' (left square bracket). Any attributes, lines of data that are mis-formed (do not have the proper number of columns), or other constructs that occur between the last line of the table and the first keyword are ignored.

*Note: For consistency, it is recommended that the user terminate a table with a keyword of the form [End <table name>].*

Except for multi-dimensional tables (next section), there must be only one table with a particular name.

Following is an example of a table that includes all the optional attributes:

```
[Lookup Table 1]
Order = 1
Below = 1
Above = 1
Resistance = 50
Volts I(typ) I(min) I(max)
0    0    0    0
1    1m   0.9m 1.1m
2    2m   1.8m 2.2m
3    3m   2.7m 3.3m
[End Lookup Table 1]
```

Here is the same data, placed in a table that uses only the required parts of a table.

```
[Lookup Table 1]
0    0    0    0
1    1m   0.9m 1.1m
2    2m   1.8m 2.2m
3    3m   2.7m 3.3m
```

---

### Referencing a table in a [Define] section

To reference a table in an expression, reference it by name, followed by the lookup value in parentheses.

Example:

```
[Define Resistor] non-linear (a1 a2)
resistor Rx (a1 a2) I = [IVcurve](V(a1,a2))
[End Resistor] non-linear
```

The table 'IVcurve' gives the value of resistor 'Rx' where the value of the argument 'V' (in parentheses) is used as an index into column zero of that table.

The corresponding data section for the above [Define] would be as follows:

```
[Resistor] R123
Resistor_type non-linear

[IVcurve]
Volts I(typ)
-5 -0.1
-2 -0.05
0 0.0
2 0.05
5 0.1

[End Resistor]
```

Here this resistor's I-V characteristics are defined by the table [IVcurve].

## 9.4 Multi-dimensional tables

---

### Abstract

This section describes making multi-dimensional tables.

---

### Description

Multi-dimensional tables are a collection of single tables that use name=value pairs at the beginning of each table to create the additional dimension. Multi-dimensional tables are used to describe a value that might be both nonlinear and time dependent. They are also used to describe multi-terminal components where the dependent variable is nonlinear in relationship to both ports.

---

### Constructing Multi-dimensional tables (data section usage)

Multi-dimensional tables are defined as a group of numeric key-value tables that have the same table name. Multi-dimensional tables are constructed using the same rules as the single dimensional tables, except that all tables in the group are required to have at least one name=value pair in common.

Following is an example of how the multi-dimensional table [IVcurve] is created. Note that each table includes the name-value pair temperature=<value>.

```
[IVcurve]
Temp = 25
Volts I(typ)
-5  -0.1
-2  -0.05
0   0.0
2   0.05
5   0.1
```

```
[IVcurve]
Temp = 0
Volts I(typ)
-5  -0.2
-2  -0.05
0   0.0
2   0.05
5   0.2
```

```
[IVcurve]
Temp = 100
Volts I(typ)
-5  -0.3
-2  -0.05
0   0.0
2   0.05
5   0.3
```

---

### IBIS 3.2 compatibility

Multi-dimensional tables are used for the following keywords in the [Model] section:

- \* Waveform
- Series MOSFET

---

### Referencing a multi-dimensional table in a [Define] section

As with single numeric key-value tables, multi-dimensional tables are referenced by name, followed by a two or more lookup values, separated by commas, in parentheses.

Example:

```
[Define Resistor] non-linear (a1 a2)
resistor Rx (a1 a2) I = [IVcurve](V(a1,a2), temperature)
[End Resistor] non-linear
```

The first (unnamed) argument in parentheses is used as an index into column 0 of one of the table in the group. The named argument (i.e. temperature=T) is used to select which table of the group the first argument indexes into.

Instead of the named attribute, a "\*" may be used to indicate that multiple tables with the same name and different attributes are allowed, and that it is handled as a special case. An example of this usage is the driver waveform tables found in IBIS 3.2.

## 9.5 Scopes

-----  
Abstract

A scope is a named set of data delimited by keywords in brackets.

-----  
Description

When constructing an IBIS data section, a portion of the data can be set aside in a separate scope. A scope is a set of data that can be referred to and operated on by name. This is useful for grouping set of data so they can be operated on by a "select" or other macro language statement. Also, additional attributes in a table are interpreted as being in a scope.

-----  
Constructing scopes (data section usage)

A scope is constructed as follows:

A scope begins with a scope header in the form of a traditional IBIS keyword in square brackets; i.e. [*scope name*]

*Note: While not required, it is recommended that the <scope name> begin with the literal 'Begin'.*

The body of a scope contains other symbols, including string variables, scalars, tables, or other scopes. Scopes may be nested.

Any line that is not a proper member of the scope terminates the scope. Specifically, the occurrence of the next keyword terminates a scope.

*Note: While not required, it is recommended that a scope be terminated with a keyword of the form*  
*[End <scope name>]*

The attributes (behavior attributes or name=value pairs) in a table are accessible as if the table were a scope.

A case statement in the macro language, if selected, brings a scope with a matching name into the enclosing scope.

-----  
 IBIS 3.2 compatibility

Scopes are used in the basic sense for [Model\_Spec] and [Submodel\_Spec]. Select and case statements, and the corresponding use of scopes, are used for the selection associated with the Series\_switch model\_type. Attributes of a table being treated as a scope are used to describe a test load circuit in [Falling\_Waveform] and [Rising\_Waveform].

-----  
 Referencing a scope in a [Define] section.

The macro languages access data in a scope by prefixing the data with the scope name. For example, a reference to “[Model\_Spec] Vmeas” in the macro language accesses the variable Vmeas in the [Model\_Spec] scope.

Scopes are also used in the ‘case’ portions of a select statement. In the example below, the user defines the value of “circuit\_selector” as either [Lumped] or [Distributed]. In the data section the user then creates the corresponding data scopes [Lumped] and [Distributed]. Note that because the data section does not contain any other named scopes, any other value of circuit\_selector is illegal.

```
[Define Test] of_select (pin gnd pad gnd)
select (circuit_selector)
  case "[Lumped]"
    capacitor C1 (pin gnd) C = C_pin
    inductor L1 (pin pad) L = L_pin_pad
    capacitor C2 (pad gnd) C = C_pad
  end case
  case "[Distributed]"
    tline T1 (pin gnd pad gnd) delay = Delay z0 = impedance
  end case
end select
[End Test] of_select
```

```
[Test] TL_test
Test_type of_select
[Lumped]
  C_pin = 1.2p
  L_pin_pad = 374p
  C_pad = 1.24p
[End Lumped]
[Distributed]
  Delay = 2.5n
impedance = 53
[End Distributed]
[End Test]
```

## 9.6 Foreach arrays

-----  
 Abstract

A set of data can be described using "foreach" arrays.  
 -----

Description

A 'foreach' array is a named set of macro language statements that are operated on by a foreach construct. The macro language statements are usually some type of assignment statement with an arbitrary number of fields that are filled in by the foreach operator.

-----  
 Constructing foreach arrays (data section usage)

A foreach array is constructed as follows:

A foreach array begins with header that identifies the array. This header is in the form of a traditional IBIS keyword; i.e. [<foreach array name>]

The body of a foreach array consists of an arbitrary number of lines with an arbitrary number of fields in each line. The fields are delimited by whitespace, and can contain no whitespace within them.

A foreach array is terminated by the occurrence of the next keyword; i.e. line that begins with a left square bracket '['.

-----  
 Referencing a foreach data array in a [Define] section

As described in section 7, the data in a foreach array is referenced using a "foreach" statement:

```
foreach foo in [A_Batch_Of_Data]
```

The loop variable (foo in this example) is assigned a sequential number (1,2, ...) which can be accessed inside a macro language statement as \$foo\$. The fields on each line are accessed by number: \$0\$, \$1\$, etc. So, for the foreach array [A Batch Of Data]:

```
[A_Batch_Of_Data]
Model_A 120 slow
Model_B 176 fast
[End A_Batch_Of_Data]
```

The macro language might describe:

```
foreach iii in [A_Batch_Of_Data]
resistor R$iii$x (pin a$iii$) R = $1$
subckt X$iii$x (a$iii$ gnd) $0$
end foreach
```

This combination is equivalent to:

```
resistor R1x (pin a1) R = 120
subckt X1x (a1 gnd) Model_A
resistor R2x (pin a2) R = 176
subckt X2x (a2 gnd) Model_B
```

A foreach statement may have an "else" section which is executed when the foreach array does not exist in the data section.

---

---

Section 10

FUTURE EXTENSIONS

---

---

This section of the LRM describes extensions that reached a consensus within the IBIS-ML committee that these items are beyond the scope of the first release. Although these are not be part of the first release of the IBIS-ML specification, these extensions are expected to be included in a future release, with the syntax TBD.

This section is included here to provide information to both users and EDA vendors on anticipated extensions. There is no guarantee that these extensions will be adopted as presented here.

Expressions will be expanded to include Frequency as an allowed state variable. This will allow support of frequency-domain models and analysis.

Math operators will be expanded to include derivatives (*d\_dt*) and integrals (*integral\_t*) with respect to TIME.

Components will be allowed to depend on terminal currents as well as node voltages. Terminals must be passed in as controlling values, in a format TBD:

*controlled\_component\_name instance\_label (node list) (control node list)(control pin list symbol = expression(voltages on nodes, currents through pins)*

---

---

temporary use only

---

---

This is stuff that should be removed from the IBIS-ML spec but not lost.

#### Background:

Current EDA tools developed specifically for signal integrity simulation and analysis apply proprietary simulation algorithms to a set of well defined and understood data, such as that contained in IBIS files. The data model, in turn, is based on a standard I/O buffer model topology. In other words, it was assumed that the I/O buffers used for digital logic can be abstracted to a set of controlled voltage and current sources connected in a standard manner, where components can be removed but not added. Given this fixed model, the user needs only to supply specific data for each I/O buffer. This data consists primarily of the DC I/V (output voltage vs. current) characteristics, plus V/T (output voltage vs. time) waveforms that describe an output's transition from one state to another. In addition, other "data book" type information is included in the format specification, to allow inclusion of information needed for system level timing analysis and error checking.

The original IBIS specification was the first industry standard method for electronically transporting I/O buffer modeling data. However, while the original IBIS specification has obtained wide industry acceptance, it has become increasingly clear that the original paradigm of data applied to a *fixed* model is no longer able to fully meet the needs of the industry. This is due to a number of issues, some of which are:

- The increasing complexity of I/O buffers
- The increasing use of programmable I/O buffers
- The advent of new signaling technologies
- Simulation methodologies that require describing the input-to-output relationships of a digital receiver, and finally
- The need to accurately model the effects of on-die power distribution, return paths, and pin to pin coupling on an I/O pin's behavior.

Therefore, the IBIS-ML macro language is proposed.

#### Relationship between IBIS and IBIS-ML:

One of the fundamental concepts behind IBIS is the separation of data and algorithm. I/O buffer modeling data is contained in an IBIS file proper, while (as described above) the algorithm(s) that operate on the data are encoded in a EDA tool and cannot be changed by the user.

Now, with the definition of IBIS-ML, this separation of data and algorithm becomes explicit. IBIS data files contain model data – simulation data as well as ‘data book’ specifications – while the corresponding IBIS-ML file defines the behavior and usage of models (and other objects) that are referenced in the IBIS-ML file. Note that both types of files are still considered ‘IBIS’ files and both use the ‘.ibs’ extension.

#### Commitment to Backwards Compatibility:

Support for IBIS versions is provided through the IBIS Standard Library.