Agilent Technologies

Advanced Design System 2002

RFIC Dynamic Link Library Guide

February 2002

Notice

The information contained in this document is subject to change without notice.

Agilent Technologies makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Agilent Technologies shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty

A copy of the specific warranty terms that apply to this software product is available upon request from your Agilent Technologies representative.

Restricted Rights Legend

Use, duplication or disclosure by the U. S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DoD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Agilent Technologies
395 Page Mill Road
Palo Alto, CA 94304 U.S.A.

Acknowledgments

Cadence® and Analog Artist® are registered trademarks of Cadence Design Systems Incorporated.
Design Framework II™ and Composer™ are trademarks of Cadence Design Systems Incorporated.

Contents

E ADS Simulator Input Syntax

Index

Chapter 1: Introduction

The *RFIC Dynamic Link* for Cadence enables you to simulate your Cadence designs in the *Advanced Design System* (ADS) environment. Designs entered in the Cadence Schematic and stored in the Cadence design database are represented on the ADS schematic via its symbol view. The circuits can be simulated together with arbitrary combinations of ADS system and circuit components using all the circuit simulators available in ADS.

The RFIC Dynamic Link requires an extension of the process library to support the netlister and also requires the development of model files in ADS format. This additional information is used to generate netlists in ADS format as shown in Figure 1-1.

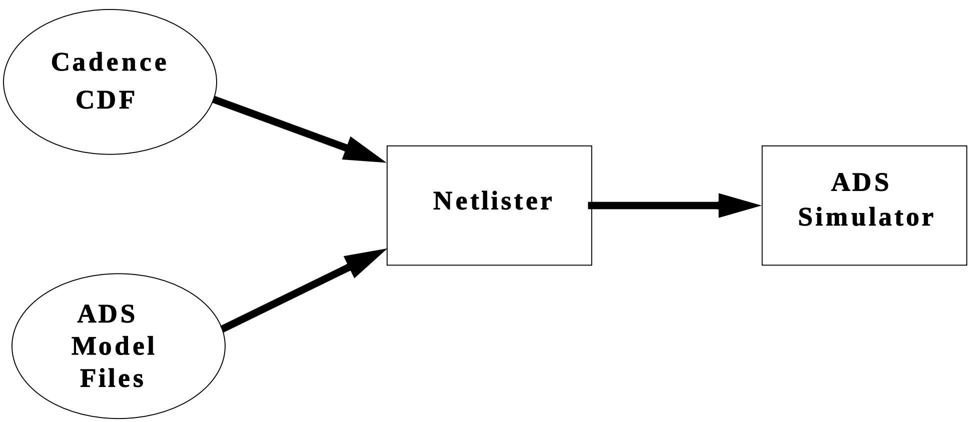


Figure 1-1. Simulation Data Flow with the RFIC Dynamic Link

Note If you are planning to use components from the *basic* and *analogLib* libraries in your designs, refer to Appendix C, Modifying the basic Library and Appendix D, Modifying the analogLib Library for additional information.

This document provides information on how to make these additions, articulated into the following two categories:

- **Creating the Netlist Interface:** This task consists of modifying the Cadence library database by adding ADS simulation information to the Component

Description Format (CDF) and creating an ADS Cellview for each library component.

- **Creating Model Files:** This is done by creating ASCII text files, formatted for ADS, that contain model parameters for each of the components.

Using Examples

Each of the above tasks is described with examples. The Dynamic Link includes a modified version of the *analogLib* library installed under *$HPEESOF_DIR/idf/cdslib/4.4.** which is used in the examples. If you do not have write access to this directory or do not want to overwrite it, make a copy of the directory first as follows:

```
cd $HPEESOF_DIR/idf/cdslib/4.4.*
find analogLib -depth -print | cpio -pd <mydir>
```

If you make a copy of the library (recommended), ensure that you edit your *cds.lib* file to point to your own copy of *analogLib* instead of to the original installed version.

Intended Audience

The information contained in this manual applies to EDA engineers and managers responsible for creating and maintaining process libraries who:

- would like to implement a design flow based on the integration of ADS and Cadence DFII using the RFIC Dynamic Link.
- have an existing Cadence component library which supports at least one commercially available SPICE simulator.
- are familiar with the Cadence library structure and Component Description Format (CDF).

If you are familiar with the topics above, you can successfully complete the library modification using the information contained in this manual.

The following rules apply to this guide

- Wherever a shell variable is set, the Korn shell syntax is presented.
- Unless otherwise mentioned, assume case sensitivity.

- If you don’t understand a particular term or acronym, refer to the Glossary in the *RFIC Dynamic Link User’s Guide*.
- For information on the ADS Cadence Menu and the Cadence AtrtistUtilities menu, refer to the “*Command Reference*” in Appendix A of the *RFIC Dynamic Link User’s Guide*.

Chapter 2: Getting ADS Device Parameter Information

This chapter describes how to obtain parameter information for devices supported by Advanced Design System (ADS). The parameter information is needed to complete the tasks outlined in subsequent chapters.

The ADS Simulator provides helpful information on netlist and model formatting via a terminal window. To use the ADS Simulator for this purpose, ensure that your environment has been configured for use with Dynamic Link. For more information on setting up your environment, refer to "*Administrative Tasks*" in chapter 2 of the "*RFIC Dynamic Link User's Guide*".

Listing Available Devices

This section describes how to use the *hpeesofsim* command to list available devices. The hpeesofsim command uses shared libraries that are set in the $HPEESOF_DIR/bin/bootscript.sh script. Before attempting to use the hpeesofsim command, you should source the *bootscript.sh* file using one of the following commands:

. $HPEESOF_DIR/bin/bootscript.sh	(If using the Korn shell)
sh; . $HPEESOF_DIR/bin/bootscript.sh	(If using the C shell)

Note The above commands are only necessary if *SHLIB_PATH* for HP-UX, *LD_LIBRARY_PATH* for SunOS, or *LIBPATH* for AIX does not include the shared libraries required to run hpeesofsim.

In a terminal window, enter:

hpeesofsim -help

A list of *Available devices and analyses* are displayed.

Getting Device Parameters

This section describes how to use the *hpeesofsim* command to obtain parameter information for a specified device. From a terminal window, enter:

hpeesofsim -help <*device_name*>

where <*device_name*> is derived using the procedure described in "Listing Available Devices" on page 2-1.

Note All device names are case sensitive. Use the *hpeesofsim -help* command to verify the correct case and spelling.

Viewing Device Output

The output of the ADS Simulator help for a specific device is a generated list of instance and model information. The output can be divided into four parts; the *Instance Statement*, the *List of Instance Parameters*, the *Model Statement* and the *List of Model Parameters*.

The examples below show the simulator output for a Bipolar Junction Transistor (BJT). To view the entire list of device parameters in a terminal window, enter:

hpeesofsim -help BJT

1. **Instance Statement -** The first section of the output produces the netlist instance statement format for the device.

   Netlist instance statement format:

   ModelName [:Name] collector base emitter ... <parameter=value> ... ; (device)

   For more information, refer to "Instance Statements" on page E-10 in Appendix E.

2. **List of Instance Parameters -** The second section contains the list of instance parameters that can be netlisted in the instance statement.

   List of available instance parameters:

```
Parameters:
    Area                smorr Junction area factor.
    Region              s---i DC operating region, 0=off, 1=on, 2=rev, 3=sat.
    Temp  (C)           smorr Device operating temperature.
```

```
    Gbe  (Siemens)   ---rr Small Signal Base Emitter Conductance.
    Cbe  (F)         ---rr Small Signal Base Emitter Capacitance.
    Gb  (Siemens)    ---rr Small Signal External Base Conductance.
    Cbc  (F)         ---rr Small Signal Internal Base Collector Capacitance.
    Cbcx  (F)        ---rr Small Signal External Base Collector Capacitance.
    Ccs  (F)         ---rr Small Signal Collector to Substrate Capacitance.
    dQbe_dVbc  (F)   ---rr Small Signal Vbc To Qbe Transcapacitance.
    dIce_dVbe  (Siemens) ---rr Small Signal Forward Transconductance gm.
    dIce_dVbc  (Siemens) ---rr Small Signal Reverse Transconductance gmr.
    dIbe_dVbc  (Siemens) ---rr Small Signal Reverse Transconductance gmr.
    dIbx_dVbe  (Siemens) ---rr External Base Transconductance dIbx_dVbe.
    dIbx_dVbc  (Siemens) ---rr External Base Transconductance dIbx_dVbc.
    NPN              s---b NPN bipolar transistor.
    PNP              s---b PNP bipolar transistor.
    Mode             s---i Nonlinear spectral model on/off.
    Noise            s---b Noise generation on/off.
```

Example of an instance statement containing some instance parameters:

NPN:Q1 c b e s Area=10 Region=1

3. **Model Statement -** The third section contains the device model statement format:

   model ModelName BJT <parameter=value> ...

   For more information, refer to “Model Statements” on page E-11 in Appendix E.

4. **List of Model Parameters -** The last section contains the model parameter information used to build the ASCII model file.

Note The use of ellipse (...) in the following output format indicates that some of the information has not been shown for conciseness.

List of available model parameters:

```
model Parameters:
    NPN              s---b NPN bipolar transistor.
    PNP              s---b PNP bipolar transistor.
    Is  (A)          smorr Saturation current.
    Js  (A)          smorr Saturation current.
    Bf               smorr Forward beta.
    Nf               smorr Forward emission coefficient.
    Vaf  (V)         smorr Forward Early voltage.
    Vbf  (V)         smorr Forward Early voltage.
    ...
    wBvbe  (V)       s--rr Base-emitter reverse breakdown voltage (warning).
```

```
   wBvbc  (V)      s--rr Base-collector reverse breakdown voltage (warning).
    wVbcfwd  (V)     s--rr Base-collector forward bias (warning).
    wIbmax  (A)      s--rr Maximum base current (warning).
    wIcmax  (A)      s--rr Maximum collector current (warning).
    wPmax  (W)       s--rr Maximum power dissipation (warning).
    Approxqb         s---b use the approximation for Qb vs Early voltage.
    Lateral          s---b Lateral substrate geometry.
    Null             s---- Has no effect.
```

Example of Model Statement containing some model parameters (note the use of the backslash (\) character):

```
model npn BJT NPN=yes Is=4.598E-16 Bf=175 Nf=0.9904 Vaf=22 Ikf=0.8 \
 Ise=1.548E-14 Ne=1.703 Br=76.1 Nr=0.9952 Var=2.1 \
 Ikr=0.02059 Isc=3.395E-16 Nc=1.13 Rb=8 Irb=8E-05 \
 Rbm=3 Re=0.45 Rc=6 Xtb=0 Eg=1.11 Xti=3 Cje=8.7E-13 \
 Vje=0.905 Mje=0.389 Cjc=3.6E-13 Vjc=0.4907 Mjc=0.2198 \
 Xcjc=0.43 Tf=1e-11 Xtf=50 Vtf=test(AAA) Itf=0.32 Ptf=32 \
 Tr=1E-09 Fc=0.6
```

In the previous definition, the parameter attributes have the following interpretation:

field 1: settable

s = settable
S = settable and required

field 2: modifiable

m = modifiable

field 3: optimizable

o = optimizable

field 4: readable

r = readable

field 5: type

b = boolean
i = integer
r = real number
c = complex number
d = device instance
s = character string

For more information on parameter attributes, refer to Table 2-1.

Table 2-1. Model Parameter Attribute Definitions

Attribute	Meaning	Example
settable	Can be defined in the instance or model statement. Most parameters are settable, there are a few cases where a parameter is calculated internally and could be used either in an equation or sent to the dataset via the OutVar parameter on the simulation component. The parameter must have its full address.	Gbe (Small signal Base-Emitter Conductance) in the BJT model can be sent to the dataset by setting `OutVar=”MySubCkt.X1.Gbe”` on the simulation component.
required	Has no default value; must be set to some value, otherwise the simulator will return an error.	
modifiable	The parameter value can be swept in simulation.	
optimizable	The parameter value can be optimized.	
readable	Can be queried for value in simulation using the OutVar parameter. See settable.	
boolean	Valid values are 1, 0, True, and False.	
integer	The maximum value allowed for an integer type is 32767, values between 32767 and 2147483646 are still valid, but will be netlisted as real numbers. In some cases the value of a parameter is restricted to a certain number of legal values.	The Region parameter in the BJT model is defined as integer but the only valid values are 0, 1, 2, and 3.
real number	The maximum value allowed is 1.79769313486231e308+.	
complex number	The maximum value allowed for the real and imaginary parts is 1.79769313486231e308+.	

Table 2-1. Model Parameter Attribute Definitions

Attribute	Meaning	Example
device instance	The parameter value must be set to the name of one of the instances present in the circuit.	The mutual inductance component (Mutual), where the parameters Inductor1 and Inductor2 are defined by instance names of inductors present in the circuit or by a variable pointing to the instance names. Inductor1=”L1” or Inductor1=Xyz where Xyz=”L1”
character string	Used typically for file names. Must be in double quotes.	Filename=”MyFileName”

Chapter 3: Creating the Netlist Interface

This chapter describes how to modify the Cadence library database. This includes creating a new *ads* symbol view for each library component as well as adding an ADS simulation information section to the Component Description Format (CDF). This procedure can be divided into the following tasks:

- Creating the *ads* Symbol View for a component
- Modifying the CDF for a component
    - Getting existing CDF information for a component
    - Editing the CDF File contents
    - Loading the modified CDF file
- Modifying the component netlisting function(s)

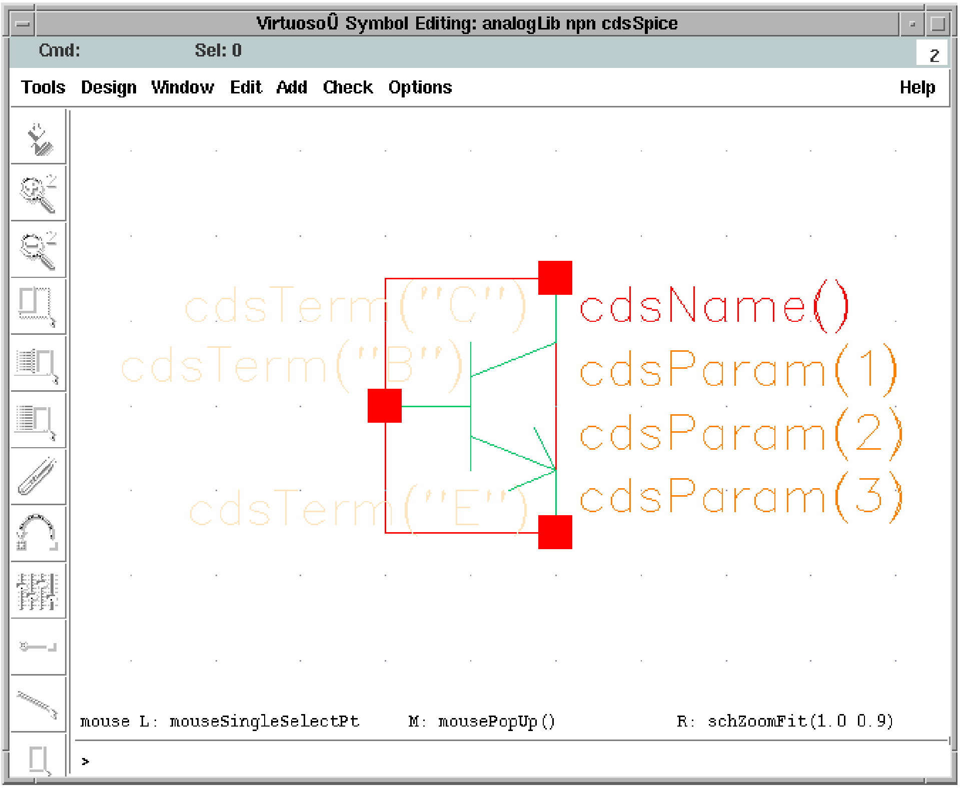
Note While the procedure for modifying the *analogLib npn* component is described, this same procedure can be applied to most any library component.

Creating the *ads* Symbol View for a Component

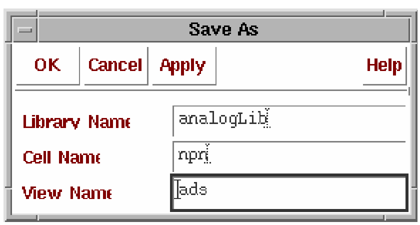
Each primitive component requires an *ads* symbol view (or *stop* view) so that the netlister knows where in the design hierarchy stops expanding the netlist. The *ads* symbol view also functions as an instance parameter template.

To create the *ads* view:

1. From the Cadence CIW, choose **File > Open** to open an existing symbol view (for example, the *cdsSpice* view) of a cell such as the analogLib *npn* cell.



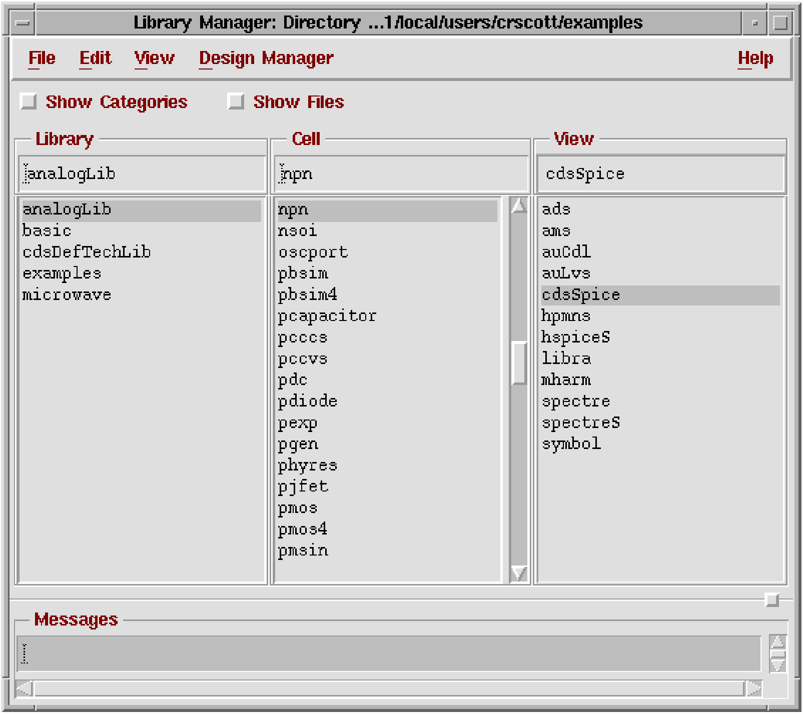
2. Choose **Design** > **Save As.** The *Save As* dialog box appears.



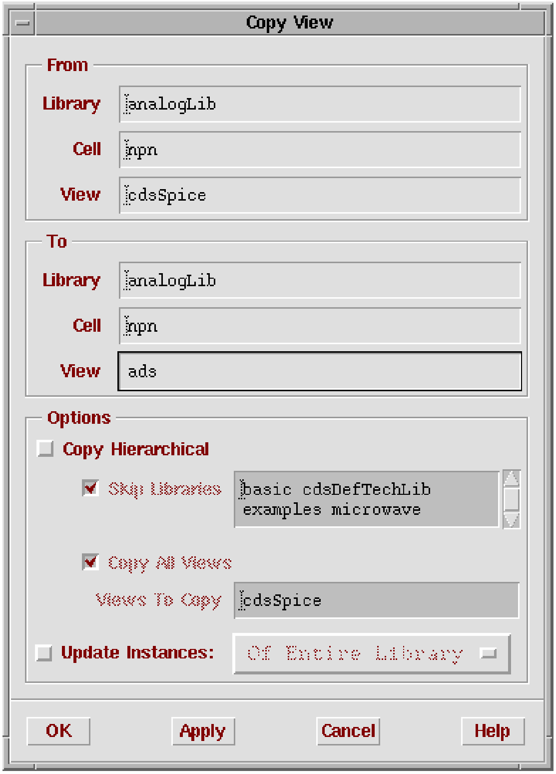
3. In the *Save As* dialog box, change the *View Name* field to *ads* and click **OK**. This creates the *ads* view in the analogLib database for the npn cell.

Alternatively, you can use the following procedure:

1. In the Cadence CIW, choose **Tools** > **Library Manager**. The Library Manager form appears.



2. In the *Library Manager* form, choose **Edit** > **Copy**. The *Copy View* form appears.



3. In the *To* section of the *Copy View* dialog box, enter *ads* in the *View* field. Ensure that all other pertinent information is correct, then click **OK**.

Modifying the Component Description Format

To modify the Component Description Format (CDF) information for a particular library component, you need the following information:

- A list of ADS instance parameters for the component. For more information, refer to “Getting Device Parameters” on page 2-2.
- The existing CDF information for the component

Getting Existing CDF Information for a Component

Although there's more than one way to obtain the CDF for a component, the most reliable way is to output the existing component CDF to a text file using the SKILL command, *cdfDump*, in the Cadence CIW window. For example:

cdfDump("analogLib" "/tmp/npn.cdf" ?cellName "npn")

Editing the CDF File

Edit the CDF information (see *Cadence Component Description Format User's Guide)* text file to make modifications (see description of the CDF files contents below). Example:

vi /tmp/npn.cdf

The CDF file consists of two main parts. The first part defines the generic *parameters* used, for example, *width* and *length*. These parameter definitions are shared by all the supported simulators under Analog Artist. The second part, known as the simulation information (*simInfo*) section, details how some subset of these parameters apply to each different simulator. This section determines how each component instance is netlisted and how its model arguments and model parameter values are output in the netlist. The simInfo sub-section of primary interest here is the *ads* siminfo sub-section, which needs to be created in order for the component to be supported by RFIC Dynamic Link.

Example CDF File

The actual CDF file may resemble the following. For conciseness only a few of the CDF parameter definitions and siminfo sub-sections have been shown here and this file was obtained as outlined in the previous step. The *ads Simulation Information* sub-section is shown highlighted.

```
/****************************************************/
 LIBRARY = "analogLib"
 CELL    = "npn"
/****************************************************/

let( ( libId cellId cdfId )
    unless( cellId = ddGetObj( LIBRARY CELL )
        error( "Could not get cell %s." CELL )
    )
    when( cdfId = cdfGetBaseCellCDF( cellId )
        cdfDeleteCDF( cdfId )
```

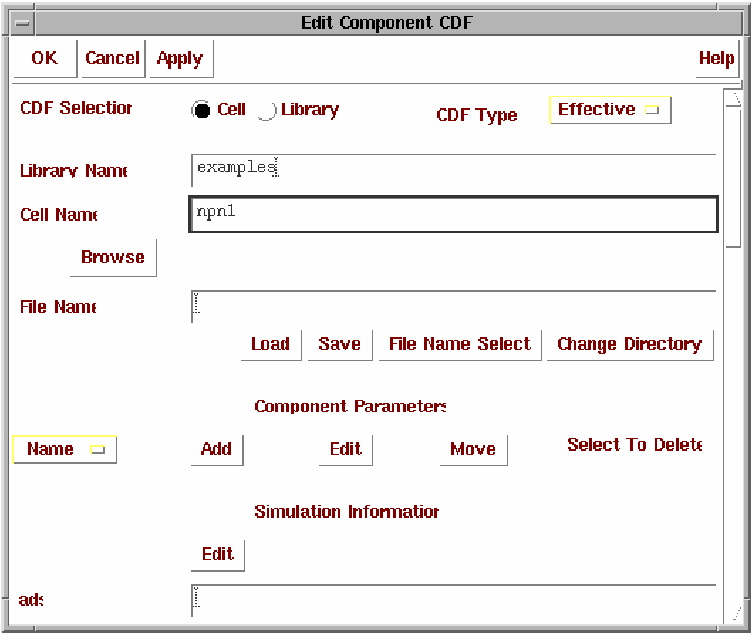
```
    )
    cdfId  = cdfCreateBaseCellCDF( cellId )

    ;;; Parameters
    cdfCreateParam( cdfId
        ?name           "model"
        ?prompt         "Model name"
        ?defValue       ""
        ?type           "string"
        ?display        "artParameterInToolDisplay('model)"
        ?parseAsCEL     "yes"
...
;;; Simulator Information
    cdfId->simInfo = list( nil )
    cdfId->simInfo->ads = '( nil
        termMapping       nil
        netlistProcedure  IdfDevPrim
        instParameters    (Area Region Temp Mode Noise)
        otherParameters   (model bn)
        propMapping       (nil Area area Region region)
        typeMapping       (nil model model)
        componentName     (expr iPar('model))
        termOrder         (C B E progn(bn))
        current           port
        namePrefix        "Q"
    )

    ...
```

Using the CDF Editor

An alternative method for editing the component CDF is by using the CDF editor. From the CIW, choose **Tools** > **CDF** > **Edit.** A dialog box enabling you to create or modify a cell's CDF information appears.

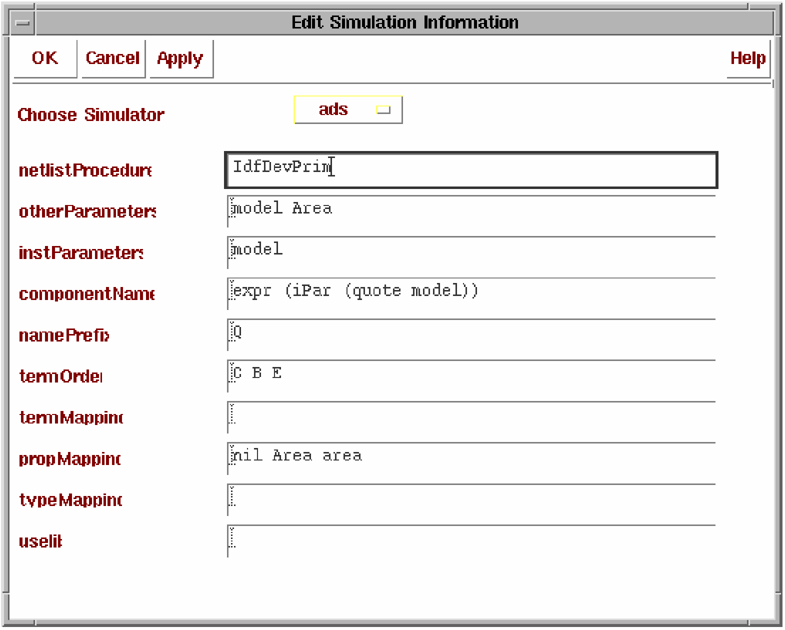


In the dialog box, add or modify the desired information. Ensure the CDF Type is set to **Base**.

Note To save CDF Edit dialog box changes, you must edit the base-level CDF and have write permission to the library.

In the *Simulation Information* section of the *Edit Component CDF* dialog box, click **Edit** to view the simInfo.

An *Edit Simulation Information* dialog box appears.



Note While the CDF *Edit Simulation Information* form may be used to edit the CDF, it is more useful to verify what is in the CDF database. Using *cdfDump()* and a text editor is more reliable for editing the CDF.

Adding CDF Simulation Information for ADS

A detailed explanation of the CDF information fields is provided in the references. However, in addition, the following applies to RFIC Dynamic Link/ADS:

- **netlistProcedure:** Use the built-in netlisting functions *IdfDevPrim* for devices requiring models (e.g., *npn, nmos*), *IdfCompPrim* for devices for which a model is not required or is optional (e.g., *cap, res*) and *IdfSubcktCall* for subcircuits.
- **otherParameters:** These are special parameters that apply to the component instance but are NOT netlisted as instance parameters (e.g., *model, bn*). These parameters appear in the *Edit Object Properties Form and the CDF*

Edit Form and are output to the netlist only if they have a value. If the value of any of these parameters is required to be netlisted (e.g., *model* value for a transistor) it should be given a value or default value (*defValue* field) in the CDF parameter definition section, otherwise the ADS simulator reports an error.

- **instParameters:** This is a list of all parameters that are netlisted as instance parameters of this component, in the form *name=value*, such as *L, W*. These parameters appear in the *Edit Object Properties Form and the CDF Edit Form and* are output to the netlist only if they have a value. If the value of any of these parameters is required to be netlisted (for example, *R* value for a resistor) it should be given a value or default value (*defValue* field) in the CDF parameter definition section, otherwise the ADS simulator reports an error.
- **modelArguments:** ADS does not support passing arguments directly to the model using this field. To pass parameters to a model it is necessary to implement the model as a subnetwork, include a model card in the subnetwork and pass parameters to the subnetwork using the *instParameters* field. So always leave out this field or set it to *nil*.
- **macroArguments:** This field is needed to pass parameters to subnetwork instances. For primitive devices leave this field blank or set it to *nil*.
- **componentName:** The content of this field is netlisted as the component name of the instance. For devices using models the component name is the name of the model. The *componentName* field may be set to an *Analog Expression Language* (AEL) expression, e.g., *expr(iPar('model))* for an *npn*. The file naming convention is <*model*>.<*suffix*> and can be any name you choose (e.g. *npn1.ads*). In the *Model name* field of the Edit Object Properties form, enter the model name. The RFIC Dynamic Link configuration file defined by IDF_CONFIG_FILE (default *idf.cfg*) specifies the *suffix* and also the *search path* (4.4.3 only) for the model file(s). For Cadence versions 4.4.5 and 4.4.6, the *Netlist File Include* component is used to locate model files. This enables the netlister to determine which model file to include in the netlist when it outputs a given instance.
- **termOrder:** This field specifies the order in which the terminals are netlisted. This information is obtained for each ADS component by entering:

  hpeesofsim –help <*device_name*>
- **termMapping:** This field defines the mapping between the pins/terminals in the schematic/symbol and the currents in the DC PSF file (see Figure 3-1). This

mapping is used to back annotate DC simulation results for currents to the schematic. Node voltages are annotated based on the node name, not the pin name, so this field has no effect on voltage annotation.

```
"I1.net8" "node" 1.450000
"gnd!" "node" 0.000000
"I1.I0:P1" "source" -0.000018
"I1.I0:P2" "source" 0.000385
"I1.I0:P3" "source" -0.000367
```

Figure 3-1. Sample of DC PSF File

The ADS simulator itself does not keep track of the pin names for devices. ADS only tracks what the pin ordering for a device was. The mapping itself must be constructed by looking at the *termOrder* field. Whatever pin is first in the *termOrder* field will then be pin 1 for the ADS simulator, the second terminal is pin 2, and so on. Figure 3-2 shows the simInfo for the analogLib *npn* component. The terminal order is listed as *C B E S*. This means that the first terminal is *C*. It needs to be mapped to terminal 1 for the ADS simulator results. In keeping with ADS convention, terminal 1 is listed as *P1*. The colon character is a delimiter character, and must be placed in the mapping. The proper mapping for *C* is thus *:P1*. When current annotations are done and the instances *C* pin is encountered, it will then look for a current source named *P1*. If the instance was in subcircuit I1, and is named I0, when pin *C* is encountered, the PSF file will be checked for *I1.I0:P1*. Looking at the PSF in Figure 3-1, we can see this would result in the value -0.000018 being annotated to the schematic. Continuing through the list, *B* is the second terminal, and is mapped to *:P2*, *E* is the third terminal and is mapped to *:P3*, and *S* is the fourth terminal and mapped to *:P4*.

```
cdfId->simInfo->ads = ‘ ( nil
     netlistProcedure   IdfDevPrim
     otherParameters    (model)
     instParameters     (Area Region Temp Mode Noise)
     componentName      (expr iPar( ‘ model))
     termOrder          (C B E S)
     termMapping        (nil C ":P1" B ":P2" E ":P3" S ":P4")
     propMapping        (nil Area area Region region)
     namePrefix          " "
     typeMapping         nil
     uselib              nil
)
```

Figure 3-2. ADS simInfo for npn device with termMapping field set

For Bi-directional elements, it turns out that the ADS simulator will only output a single current value. A case in point is the ADS *R* element (an ideal resistor). In order to annotate both pins, it becomes necessary to specify that one pin is the negative of the other pin (in other words, current enters through one pin (+), and leaves through the other pin (-)). This mapping can be achieved by placing a key word of *minus.* in front of the mapped pin name. Figure 3-3 displays the analogLib *res* simInfo, where bi-directional mapping has been done. The *termOrder* field is *PLUS MINUS*. *PLUS* has been mapped to *:P1*, as would be expected. However, *MINUS* has been mapped to *:minus.P1*. This specifies to the annotation code that, when *MINUS* is encountered, the current for the positive terminal should be retrieved, and it’s value should be multiplied by -1.

```
cdfId->simInfo->ads = ‘ ( nil
     netlistProcedure   IdfCompPrim
     otherParameters    (wPmax wImax Model)
     instParameters     (R Temp Tnom TC1 TC2 Width Length Noise)
     componentName       R
     termOrder          (PLUS MINUS)
     termMapping        (nil PLUS ":P1" MINUS ":minus.P1")
     propMapping        (nil R r Tnom tnom TC1 tc1 TC2 tc2 Width w \
Length l Model model Noise isnoisy)
     namePrefix          " "
     typeMapping         nil
     uselib              nil
)
```

Figure 3-3. ADS simInfo for res device showing the minus keyword in termMapping

The termMapping field does not need to be set for hierarchical devices. Hierarchical circuits will descend into the hierarchy and retrieve the currents of all devices attached to a port, and add them together. This does make it critical that the *minus* key word be used properly on bi-directional devices. If *minus* is not used, when the currents are added up at a port, the value that is annotated will not be correct. Regrettably, even if a termMapping is set up for a hierarchical device, and an entry exists in the PSF file, it will still not be used, the internal Cadence code will always descend into the hierarchy and add up the values.

Note Voltages and Currents will only be annotated on pins that have an associated cdsTerm. This is true for primitive devices as well as for hierarchical subcircuits.

- **propMapping:** This allows parameter definitions to be reused or shared even though they have different names (for use by different simulators) and acts as an aliasing mechanism. For instance, the parameter named *Area* used by ADS is mapped to *area* which most other simulators use. In fields like *instParameters* and *otherParameters*, the simulator-specific name (e.g., *Area*) should be used.
- **namePrefix:** Used as a prefix for instance names.

- **typeMapping:** This field is used to call a built-in SKILL function to netlist certain types of parameters, whenever they are given a value. e.g.,mapping a property to type *substrate* for *microwave* library components will cause the *IdfPrintSubstrate()* function to be called whenever *Subst* has a value:

  ```
  propMapping(nil Subst subName)
  typeMapping(nil Subst substrate)
  ```

  To get a list of all such mappings, type the following in the CIW:

  ```
  asiGetNetlistOption(asiGetTool('ads) 'propTypeMapping))
  ```

The *npn* has been instantiated as shown in the figure below with the connecting wires named according to the device terminals.

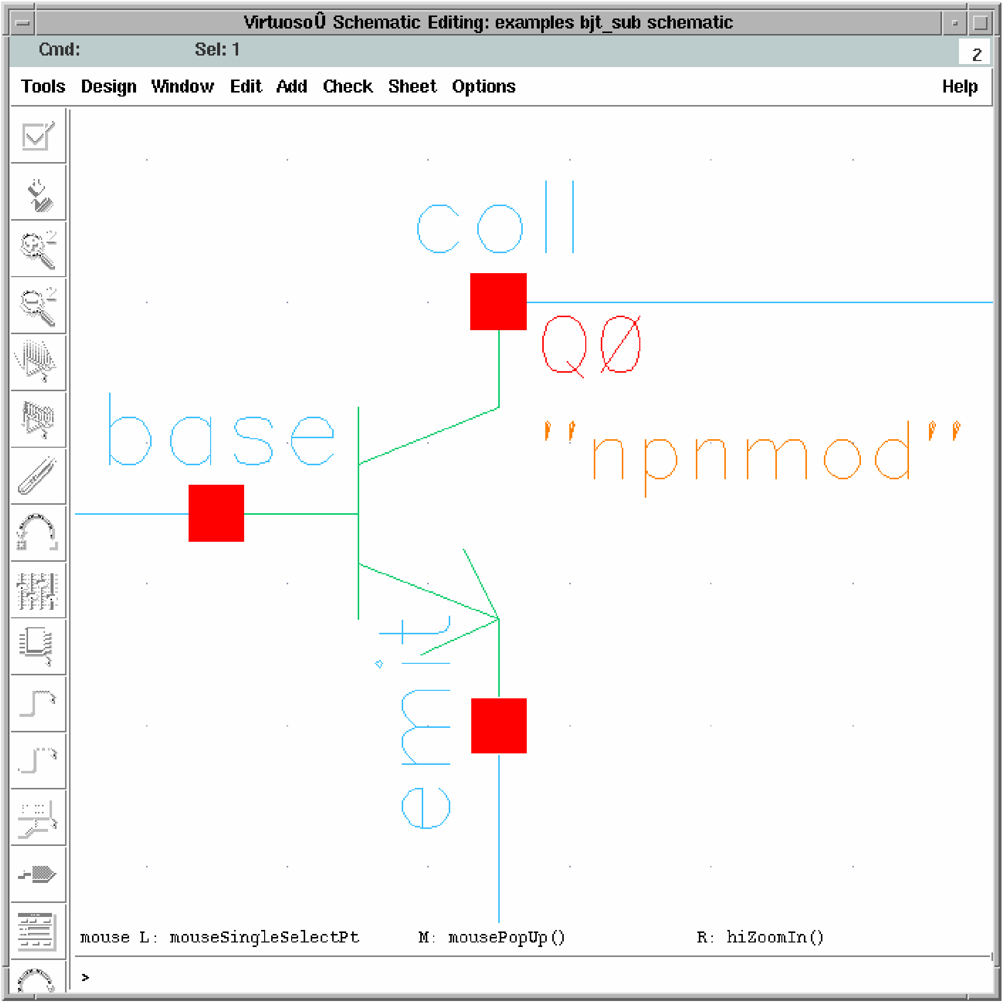
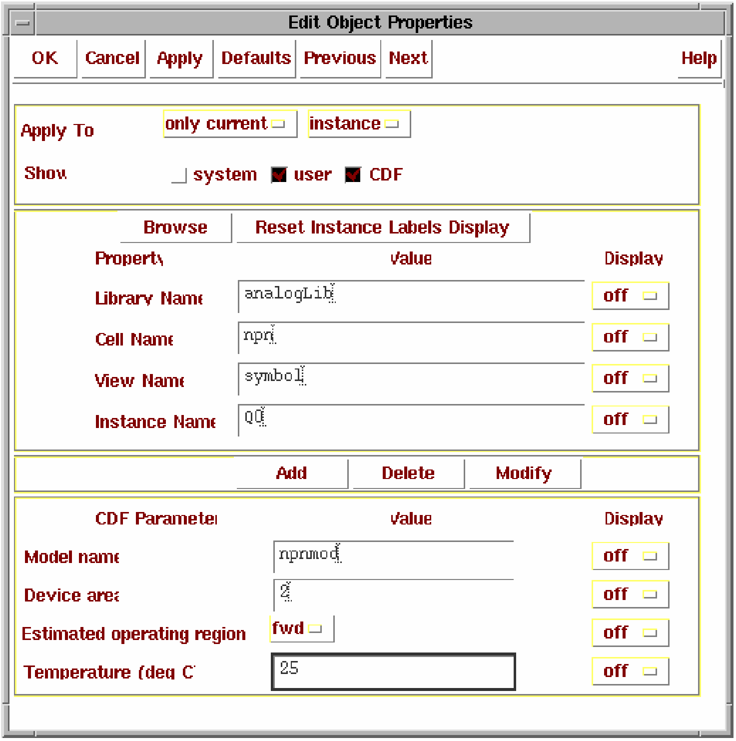


Figure 3-4. Instance of npn Component

The object parameters for this instance have been set as follows:



The instance statement on the ADS netlist corresponding to this instance will appear as follows:

```
...
npnmod:Q0 coll base emit 0 Area=2.0 Region=1 Temp=25.0
...
```

The following model file will also be appended to the netlist:

```
...
model npnmod BJT NPN=yes Is=4.598E-16 Bf=175 Nf=0.9904 Vaf=22 \
Ikf=0.8 Ise=1.548E-14 Ne=1.703 Br=76.1 Nr=0.9952 Var=2.1 \
Ikr=0.02059 Isc=3.395E-16 Nc=1.13 Rb=8 Irb=8E-05 \
Rbm=3 Re=0.45 Rc=6 Xtb=0 Eg=1.11 Xti=3 Cje=8.7E-13 \
Vje=0.905 Mje=0.389 Cjc=3.6E-13 Vjc=0.4907 Mjc=0.2198 \
Xcjc=0.43 Tf=1e-11 Xtf=50 Vtf=test(AAA) Itf=0.32 Ptf=32 \
Tr=1E-09 Fc=0.6
...
```

The instance statement on the ADS netlist corresponding to this instance contains the following parameters:

- **npnmod** The netlister evaluates the expression contained in the *componentName* CDF field and in this case picks up the value of the model name property (`expr ipar('model)`). The netlister also appends the content of the *npnmod.ads* file.
- **Q0** The instance name is generated by using the contents of the *namePrefix* CDF field and appending an incremental number (i.e. Q0, Q1, Q2,...).
- **coll base emit 0** The first three entries are taken from the names of the nodes to which the device is attached (see Figure 3-4). In this case, the names have been explicitly assigned but the same applies to system generated node names. The *termOrder* field in the CDF controls the order in which the terminals are netlisted.

Note The *progn* SKILL function is no longer supported by RFIC Dynamic Link in Cadence version 4.4.5 and above.

- **Area=2.0 Region=1 Temp=25.0** The parameters *Area*, *Region* and *Temp* are listed in the *instParameters* field of the component CDF, therefore they are netlisted as instance properties if their value has been set on the instance. If the field is left blank, the parameter is not netlisted and the simulator uses the default value.
- **model npnmod** ... The netlister appends the contents of the file <*Model name*>*.ads* (if the IDF_MODEL_SUFFIX variable is set to the default value), which in this case is the model file for *npnmod*.

Additional Notes for Simulation Information Fields

- All *simInfo* parameters that apply to the *Microwave* and *hpmns* Cadence Analog Artist interfaces also apply to the *ads* simulator view. An example of such a parameter is *typeMapping*.
- When errors in the CDF file are loaded with *load* <*file*>, command errors may not be reported. If this occurs, the corresponding *ads* simulation view for the device is not created.

Loading the Modified CDF File

After modifying the CDF text file to support ADS, load the edited file from the CIW using the SKILL command, *load*. For example:

```
load "/tmp/npn.cdf"
```

This automatically updates the Cadence library database and saves the new CDF information in the database, provided you have write permissions.

Modifying the Component Netlisting Function(s)

Each simulator can use its own netlist function to write out a component instance in its own netlist format. Two built-in component-netlisting procedures are available in the RFIC Dynamic Link SKILL context:

- *IdfDevPrim* is used for components that always need a model (a transistor, for example)
- *IdfCompPrim* is used for components that may or may not need models (a resistor, for example)

You probably *won't* need to modify or replace these functions. But if you do, the SKILL code for these built-in functions is provided in:

```
$HPEESOF_DIR/idf/skill/netlistFuncs.il
```

Chapter 4: Creating Model Files

This chapter describes how to create ASCII-text process-dependent model files, formatted for ADS. These files are stored separate from the Cadence library database, in a model library directory. The netlister will simply append the model file to the final top-level ADS netlist without a syntax check. The ADS simulator requires the syntax of these files to be exact.

To build model files in ADS format, you'll need the following information:

- The basic built-in ADS component parameter information (refer to "Getting Device Parameters" on page 2-2).
- The ADS Simulator Input format information (refer to Appendix E, ADS Simulator Input Syntax).

This chapter describes the following tasks:

- Creating Process Parameter Files
- Linking the ADS Model File to a Library Component

Creating a Simple ADS Model File

Once the model parameters are known, you can create an ADS model file using an ASCII text editor. In your text editor window, type in the complete model statement in the appropriate format for the selected device as defined in part 3 of "Viewing Device Output" on page 2-2. As you build the ADS model file, be aware of the following:

- The model statement must be on a single line. Use the backslash (\) as a line continuation character.
- The instance and model parameter names are case sensitive.
- If a parameter is not specified, ADS uses a default parameter value. These values are documented in volume 1 of the ADS "Circuit Components" manual.

Example:

```
model npn BJT NPN=yes Is=4.598E-16 Bf=175 Nf=0.9904 Vaf=22 \
Ikf=0.8 Ise=1.548E-14 Ne=1.703 Br=76.1 Nr=0.9952 Var=2.1 \
Ikr=0.02059 Isc=3.395E-16 Nc=1.13 Rb=8 Irb=8E-05 \
Rbm=3 Re=0.45 Rc=6 Xtb=0 Eg=1.11 Xti=3 Cje=8.7E-13 \
Vje=0.905 Mje=0.389 Cjc=3.6E-13 Vjc=0.4907 Mjc=0.2198 \
Xcjc=0.43 Tf=1e-11 Xtf=50 Vtf=1.2 Itf=0.32 Ptf=32 \
Tr=1E-09 Fc=0.6
```

Creating a Parametric Subnetwork Model File

Device models, especially for active devices, often consist of complex combinations of primitive components such as resistors, inductors, capacitors, diodes and transistors. These model files are thus structured as subnetworks, that also allow parameters to be set on the instance and passed down the hierarchy to the subnetwork.

The syntax supported by the ADS Simulator is described in Appendix E under "Subcircuit Definitions" on page E-12

Example:

```
define npn1 ( c b e )
parameters Area=1 Region=1 Noise=1
model NPN BJT NPN=yes Is=4.598E-16 Bf=175 Nf=0.9904 Vaf=22 Ikf=0.8 \
Ise=1.548E-14 Ne=1.703 Br=76.1 Nr=0.9952 Var=2.1 \
Ikr=0.02059 Isc=3.395E-16 Nc=1.13 Rb=8 Irb=8E-05 \
Rbm=3 Re=0.45 Rc=6 Xtb=0 Eg=1.11 Xti=3 Cje=8.7E-13 \
Vje=0.905 Mje=0.389 Cjc=3.6E-13 Vjc=0.4907 Mjc=0.2198 \
Xcjc=0.43 Tf=1e-11 Xtf=50 Vtf=1.2 Itf=0.32 Ptf=32 \
Tr=1E-09 Fc=0.6
NPN:qin c b e 0
end npn1
```

Defining Instance Parameters using Expressions

Instance parameters must be defined in the *Component Parameters* section of the Cadence CDF as described in the *Cadence Component Description Format User's Guide*. RFIC Dynamic Link supports netlisting of instance parameters that contain Cadence AEL expressions, such as math operators, *iPar*, *pPar* etc.

Defining Model Parameters using Expressions

Model parameters contained in ADS model files can include expressions. The expressions can be defined by arbitrary combinations of predefined ADS functions, math operators and Boolean operators. For a list of functions and operators supported by ADS, refer to Appendix E, ADS Simulator Input Syntax.

For an expression to be correctly evaluated by ADS, both the syntax of the expression and the value of the variables used in the expression must be defined in one of the following places:

1. directly in the model file,
2. in a separate file which is included in the top level netlist,
3. in a separate file which is included in the model file, or
4. on the ADS top level schematic in a *VarEqn* block.

Note These different methods can be used in combination, with expressions defined in different places, as long as there is a single definition for each expression.

Example:

This model file for a BJT contains a model parameter, Vtf, that is defined as an expression of the variable AAA.

```
model npn BJT NPN=yes Is=4.598E-16 Bf=175 Nf=0.9904 Vaf=22 Ikf=0.8 \
 Ise=1.548E-14 Ne=1.703 Br=76.1 Nr=0.9952 Var=2.1 \
 Ikr=0.02059 Isc=3.395E-16 Nc=1.13 Rb=8 Irb=8E-05 \
 Rbm=3 Re=0.45 Rc=6 Xtb=0 Eg=1.11 Xti=3 Cje=8.7E-13 \
 Vje=0.905 Mje=0.389 Cjc=3.6E-13 Vjc=0.4907 Mjc=0.2198 \
 Xcjc=0.43 Tf=1e-11 Xtf=50 Vtf=test(AAA) Itf=0.32 Ptf=32 \
 Tr=1E-09 Fc=0.6
```

In order to simulate this model in ADS, the expression *test* needs to be defined and a value must be given to the variable *AAA*.

Assuming that:

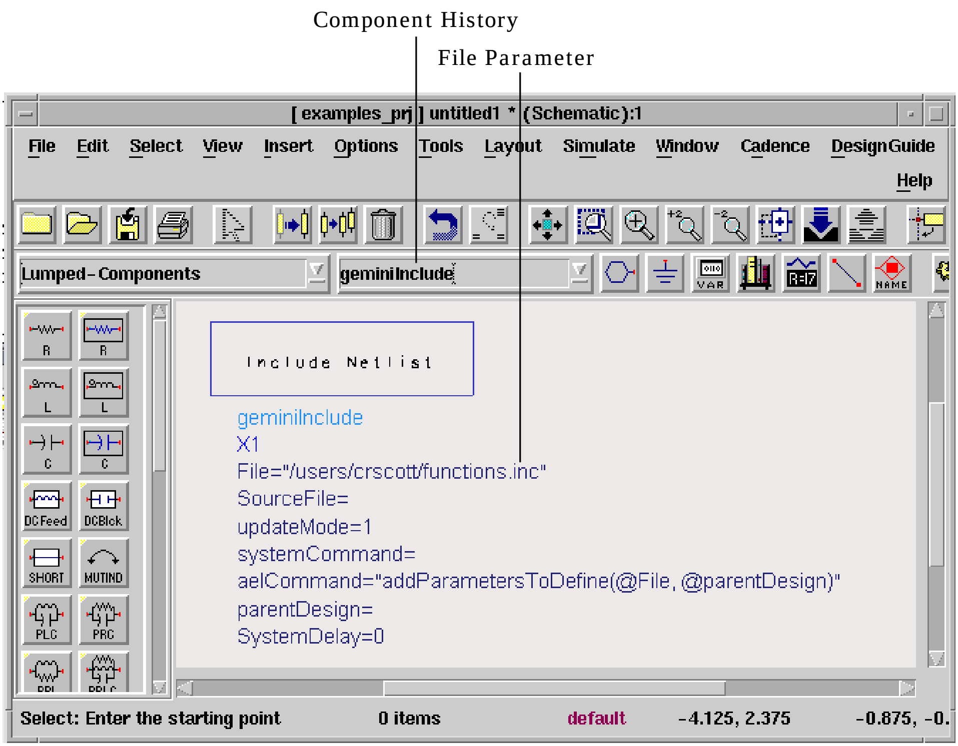
test(x)=x*1.2
AAA=1

Do one of the following:

1. Append the definition of *test* and *AAA* to the model file:

```
model npn BJT NPN=yes Is=4.598E-16 Bf=175 Nf=0.9904 Vaf=22 Ikf=0.8 \
...
 Xcjc=0.43 Tf=1e-11 Xtf=50 Vtf=test(AAA) Itf=0.32 Ptf=32 \
 Tr=1E-09 Fc=0.6
 test(x)=x*1.2
 AAA=1
```

2. Create a separate ASCII file (for example, *function.inc*) containing the definition of *test* and *AAA*. Then place a *geminiInclude* instance on the top level ADS schematic by typing *geminiInclude* (case sensitive) in the *Component History* field.



The *File* parameter should contain the full path of the ASCII file. When this component is netlisted by ADS, it generates a **#include** statement that is later replaced by the contents of the ASCII file. For more information on file inclusion, refer to Appendix E, "File Inclusion" on page E-35.

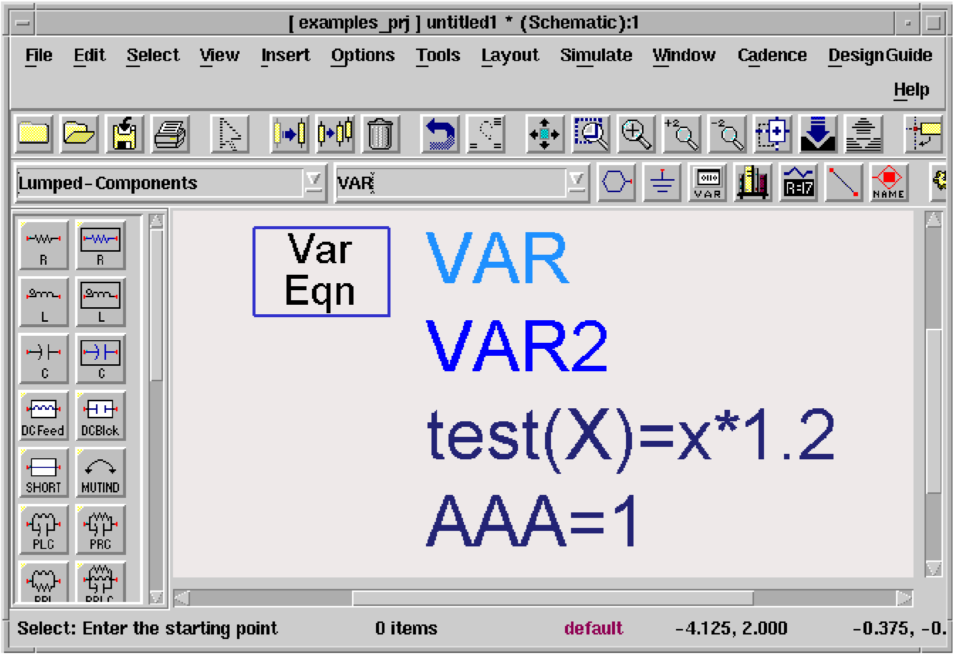
The *geminiInclude* component can thus be used to append a file containing multiple models or even the entire set of models. It can also be used to select among various files containing different sets of process parameters corresponding to different corner cases.

In a practical example, *typical.inc* could contain the process parameter values (sheet resistance, area capacitance, etc.) for the typical case, while *maximum.inc* would have definitions corresponding to the maximum case. The *geminiInclude* component can then be used to select which corner case to simulate by pointing to either *typical.inc* or *maximum.inc*.

3. Include the ASCII file with the expression definitions directly in the model file.

```
model npn BJT NPN=yes Is=4.598E-16 Bf=175 Nf=0.9904 Vaf=22 Ikf=0.8 \
...
 Xcjc=0.43 Tf=1e-11 Xtf=50 Vtf=test(AAA) Itf=0.32 Ptf=32 \
 Tr=1E-09 Fc=0.6
 #include "/users/home/functions.inc"
```

4. Use a VAR block in the ADS top level schematic that contains the expression definitions. For more information on the VAR block, refer to the "*VAR (Variables and Equations Component)*" in the ADS *Circuit Components* manual.



Note If an expression is used to define a model parameter, the argument cannot be another model parameter or an instance parameter. If the model needs to use the value of an instance parameter in the calculation of a model parameter, this requires creating a subcircuit that incorporates the model, as in the following example:

```
define npn1 ( c b e )
parameters AAA=1 Area=1 Region=1 Noise=1
model NPN BJT NPN=yes Is=4.598E-16 Bf=175 Nf=0.9904 Vaf=22 Ikf=0.8 \
Ise=1.548E-14 Ne=1.703 Br=76.1 Nr=0.9952 Var=2.1 \
Ikr=0.02059 Isc=3.395E-16 Nc=1.13 Rb=8 Irb=8E-05 \
Rbm=3 Re=0.45 Rc=6 Xtb=0 Eg=1.11 Xti=3 Cje=8.7E-13 \
Vje=0.905 Mje=0.389 Cjc=3.6E-13 Vjc=0.4907 Mjc=0.2198 \
Xcjc=0.43 Tf=1e-11 Xtf=50 Vtf=test(AAA) Itf=0.32 Ptf=32 \
Tr=1E-09 Fc=0.6
NPN:qin c b e 0
end npn1
```

Appendix A: References

The following references supplement the information in this book. All the Cadence manuals are available in Cadence *Openbook*.

[1] Cadence *Component Description Format User's Guide*

[2] Cadence *Design Framework II/Library Manager Help*

[3] Cadence *Analog Artist SKILL Reference*

[4] Cadence *SKILL Language Reference Manual*

[5] Cadence *SKILL User Guide*

[6] ADS "*Expressions, Measurements, and Simulation Data Processing*"

Appendix B: Adding *CDF/SimInfo* to a Component Library

The chapter provides information on modifying the Cadence simInfo (Simulation Information) section in a CDF (Component Description Format) file.

Using *cdfDumpAll*

The benefit of adding simulator information via *cdfDumpAll* is that you need not have numerous files containing specific simulation parameters and simInfo. Instead, all of the CDF information is compiled for you in a single ASCII file. This method is probably your best choice if you do not have source files for parameter and simInfo data for each and every simulator that a library currently supports.

Dumping the CDF for an Entire Component Library

To create and modify an ASCII file containing the entire CDF for an existing component library:

- Enter the following Skill command in the Cadence CIW:

  cdfDumpAll("*libName*" "*fileName*" ?edit t)

- In the text editor of your choice (*vi*, *emacs*, etc.), for each library cell add the *simInfo* for the new simulator a*ds* to the CDF file. In some cases, you may also need to add new CDF parameters.
- Load this file in the CIW using the command:

  load "*fileName*"

  This modifies the library database accordingly, assuming you have write permission to the library.

Dumping the CDF for Individual Components

To create and modify an ASCII file containing the CDF for an individual component:

- Enter the following Skill command in the Cadence CIW:

  cdfDump("libName" "fileName" ?cellName "cellName" ?edit t)

- In the text editor of your choice (*vi*, *emacs*, etc.), for each library cell add the *simInfo* for the new simulator a*ds* to the CDF file. In some cases, you may also need to add new CDF parameters.
- Load this file in the CIW using the command:

  load "*fileName*"

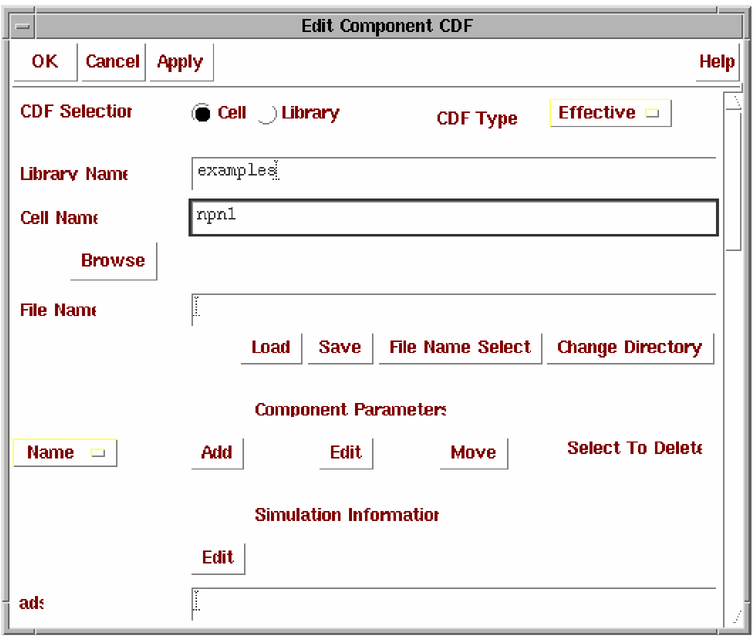
  This modifies the library database accordingly, assuming you have write permission to the library.

Using the Edit Component CDF Form

Adding CDF information via the Edit Component CDF form is the ideal method for those who are not computer programmers. It is also often the best method to use when changes to only a few cells are required.

To add new CDF information via the Edit Component CDF form:

- From the CIW, choose **Tools > CDF > Edit.** A dialog box enabling you to create or modify a cell's CDF information appears.



- In the dialog box, add or modify the desired information.

Note To save changes to the Edit Component CDF form, you must edit the base-level CDF and have write permission to the library.

For more details on using the Edit Component CDF form, refer to the *Cadence Component Description Format User's Guide* [1].

Note If you are adding a CDF entry for a new simulator, the tool filter file must reflect this before the entry appears in the dialog box's simulation information (simInfo) section. For more information, refer to the Cadence *Component Description Format User Guide.*

Appendix C: Modifying the *basic* Library

RFIC Dynamic Link requires that the *basic* library *nlpglobals* cell contains the *ads* view. A version of the *basic* library is located in

$HPEESOF_DIR/idf/cdslib/4.4.*/basic

Alternatively, you may modify your site's version of the *basic* library located in:

<*Cadence_install_dir*>/tools/dfII/etc/cdslib/basic

To do this:

- Using the Cadence Schematic window, edit the *spectre* view of cell *nlpglobals*.
- Save this view as the *ads* view.

Appendix D: Modifying the *analogLib* Library

The RFIC Dynamic Link install package includes a version of Cadence *analogLib* that has been extended to work with ADS and is located in:

$*HPEESOF_DIR*/*idf*/cdslib/4.4.*/analogLib

However, if you need to extend *your own* version of *analogLib* to work with ADS, this appendix may be useful.

To modify your version of *analogLib*:

1. Make a temporary directory called *adsLib* at the current level then change to the newly created *adsLib* directory.
2. Copy your version of *analogLib* to your current (*adsLib)* directory. Take care to use a method, such as UNIX *tar*, that will preserve the file dates and access codes.

   AnalogLib is usually located in $CDS_INST_DIR/tools/dfII/etc/cdslib/artist/

   Alternatively, you can use the UNIX copy command:

   cp -r $CDS_INST_DIR/tools/dfII/etc/cdslib/artist/analogLib .
3. Copy the official versions of some or all of the following simulator directories (usually located under *$CDS_INST_DIR/tools/dfII/src/artist/*) to your current directory.

   auCdl
   auLvs
   cdsSpice
   hpmns
   hspiceS
   libra
   microwave
   spectre
   spectreS
   spice2

   The above directories are listed in alphabetical order. Each should contain *simInfo.il* files for the respective simulators.

Note Instead of copying these directories, you may want to make symbolic links to them.

---

4. Copy the official version of the *ads* simulator directory (located under *$HPEESOF_DIR/idf/cdslib/4.4.*/artist/ads/*) to your current directory. Make your modifications to the appropriate *SKILL* files in the ads directory.
5. Create a one-line cds.lib file that defines analogLib. The content of the cds.lib file should contain:

   DEFINE analogLib ./analogLib

6. Enter the command:

   makeAnalogLib

7. Copy the newly created *analogLib* to whatever location you desire, such as:

   $CDS_INST_DIR/tools/dfII/etc/cdslib/artist/analogLib

You are now able to simulate in ADS using the modified *analogLib* library.

Using *almBuildLibrary* in a UNIX Shell Script

The Analog Artist Skill function *almBuildLibrary* compiles the simulation information for various simulators into a given library. For each such library, you will need to write a UNIX shell script that essentially starts *icms* in non-graphics mode and then runs *almBuildLibrary*.

The following is an example script for *analogLib*, a variation of which can usually be found in *<Cadence_install_dir>/tools/dfII/src/artist/analogLib/makeAnalogLib*:

```
#!/bin/csh -f

echo Building library...
/bin/rm -f CDS.log
cat << EOF > tmp.il
\i printf("Loading tmp.il...")
\i lib = "analogLib"
\i sourcePath = "."
\i simulators = '( ads auCdl auLvs cdsSpice hpmns hspiceS libra spectre \
spectreS spice2 hpmns )
\i ddGetObj( lib )
\i sstatus( writeProtect nil)
\i load("./ads/params.il")
\i load("./ads/labels.il")
\i (almBuildLibrary ?lib lib ?sourcePath sourcePath ?simulators
simulators)
\i exit()
EOF

icms -replay ./tmp.il -nograph -log ./CDS.log
```

For this example script to work, there must be:

- a copy of *analogLib* in the current directory
- a subdirectory for each of the simulators
- and each simulator directory must contain a file called *simInfo.il*.

Your directory structure should be similar to the following:

```
adsLib/
  ├── analogLib
  ├── cds.lib
  ├── makeAnalogLib
  ├── ads/
  │     ├── params.il
  │     ├── labels.il
  │     └── simInfo.il
  ├── spectre/
  │     └── simInfo.il
  └── ...
```

This procedure is documented in more detail in the *Cadence Component Description Format User's Guide* [1].

Appendix E: ADS Simulator Input Syntax

This chapter provides information related to Advanced Design System's Simulator. While this is not an all inclusive document with regards to the ADS simulator, the information provided in this chapter should help you accomplish tasks related to the RFIC Dynamic Link.

Operating System Requirements

The ADS 2002 Simulator is supported on the following platforms:

- HP-UX 10.20 or 11
- SunOS 5.6, 5.7 & 5.8 (Solaris 2.6, 7.0 & 8.0)
- AIX 4.4.3 or later
- Windows 98, 2000, and NT 4.0

Setting Environment Variables

Before running the ADS Simulator, the following environment variables must be set:

Table 4-1. ADS Simulator Required Environment Variables

Variable	UNIX Setting
HPEESOF_DIR	<*ADS_install_dir*>
PATH	$PATH:$HPEESOF_DIR/bin

To set the UNIX environment variables using the Korn Shell, add the following to your *~/.profile*.

```
export HPEESOF_DIR=<ADS_install_dir>
export PATH=$PATH:$HPEESOF_DIR/bin
```

To set the UNIX environment variables using the C Shell, add the following to your *~/.cshrc*.

```
setenv HPEESOF_DIR <ADS_install_dir>
setenv PATH $PATH:$HPEESOF_DIR/bin
```

In addition to HPEESOF_DIR and PATH, you also need to set COMPL_DIR. The COMPL_DIR variable should have the same value as HPEESOF_DIR. There are times when COMPL_DIR can be different than HPEESOF_DIR; however, the majority of users should set COMPL_DIR to be the same as HPEESOF_DIR.

Platform-Specific Variables

A platform-specific variable also needs to be set before running the ADS simulator.

HP-UX:

```
export SHLIB_PATH="$HPEESOF_DIR/hptolemy/lib.hpux10:$SHLIB_PATH"
export SHLIB_PATH="$HPEESOF_DIR/lib/hpux10:$SHLIB_PATH"
```

Solaris 5.6:

```
export LD_LIBRARY_PATH="$HPEESOF_DIR/hptolemy/lib.sun56:$LD_LIBRARY_PATH"
export LD_LIBRARY_PATH="$HPEESOF_DIR/lib/sun56:$LD_LIBRARY_PATH"
```

Solaris 5.7:

```
export LD_LIBRARY_PATH="$HPEESOF_DIR/hptolemy/lib.sun57:$LD_LIBRARY_PATH"
export LD_LIBRARY_PATH="$HPEESOF_DIR/lib/sun57:$LD_LIBRARY_PATH"
```

Solaris 5.8:

```
export LD_LIBRARY_PATH="$HPEESOF_DIR/hptolemy/lib.sun57:$LD_LIBRARY_PATH"
export LD_LIBRARY_PATH="$HPEESOF_DIR/lib/sun57:$LD_LIBRARY_PATH"
```

IBM AIX:

```
export LD_LIBRARY_PATH="$HPEESOF_DIR/hptolemy/lib.aix4:$LD_LIBRARY_PATH"
export LD_LIBRARY_PATH="$HPEESOF_DIR/lib/aix4:$LD_LIBRARY_PATH"
```

MS Windows:

```
path %HPEESOF_DIR%/hptolemy/lib.win32;%PATH%
path %HPEESOF_DIR%/lib/win32;%PATH%
```

Note The platform-specific variable information above is for those using the Korn Shell or Borne Shell. Use the appropriate equivalent command if using the C Shell.

Using the hpeesofsim Command

The ADS Simulator can be invoked using the following syntax.

Usage: hpeesofsim [-r rawfile] [inputfile]

A list of available options can be generated using the following command:

Usage: hpeesofsim -o

Codewording and Security

The ADS Simulator is a secured program that requires, at a minimum, a license for the E8881 Linear Simulator to run. Depending on the type of simulation, additional licenses may be required. For more information on codewording and security, refer to "*Setting Up Licenses on UNIX Systems*" in the ADS "*Installation on UNIX Systems*" manual.

General Syntax

In this appendix, the following typographical conventions apply:

Table 4-2. Typographic Conventions

Type Style	Used For
[. . .]	Data or character fields enclosed in brackets are optional.
italics	Names and values in italics must be supplied
bold	Words in bold are ADS simulator keywords and are also required.

The ADS Simulator Syntax

The following sections outline the basic language rules.

Field Separators

A delimiter is one or more blanks or tabs.

Continuation Characters

A statement may be continued on the next line by ending the current line with a backslash and continuing on the next line.

Name Fields

A name may have any number of letters or digits in it but must not contain any delimiters or non alphanumeric characters. The name must begin with a letter or an underscore (_).

Table 4-3. Fundamental Units

Dimension	Fundamental Unit
Frequency	Hertz
Resistance	Ohms
Conductance	Siemens
Capacitance	Farads
Inductance	Henries
Length	meters
Time	seconds
Voltage	Volts
Current	Amperes
Power	Watts
Distance	meters
Temperature	Celsius

Parameter Fields

A parameter field takes the form *name* = *valu*e, where *name* is a parameter keyword and *value* is either a numeric expression, the name of a device instance, the name of a model or a character string surrounded by double quotes. Some parameters can be indexed, in which case the name is followed by *[i]*, *[i,j]*, or *[i,j,k]*. *i*, *j*, and *k* must be integer constants or variables.

Node Names

A node name may have any number of letters or digits in it but must not contain any delimiters or non alphanumeric characters. If a node name begins with a digit, then it must consist only of digits.

Lower/Upper Case

The ADS Simulator is case sensitive.

Units and Scale Factors

An integer or floating point number may be scaled by following it with either an **e** or **E** and an integer exponent (e.g., 2.65e3, 1e-14).

An ADS Simulator parameter with a given dimension assumes its value has the corresponding units. For example, for a resistance, R=10 is assumed to be 10 Ohms. The fundamental units for the ADS Simulator are shown in Table 4-3.

A number or expression can be scaled by following it with a scale factor. A scale factor is a single word that begins with a letter or an underscore. The remaining characters, if any, consist of letters, digits, and underscores. Note that "/" cannot be used to represent "per". The value of a scale factor is resolved using the following rule: If the scale factor exactly matches one of the predefined scale-factors (Table 4-4), then use the numerical equivalent; otherwise, if the first character of the scale factor is one of the legal scale-factor prefixes (Table 4-5), the corresponding scaling is applied.

Table 4-4. Predefined Scale Factors

Scale Factor	Scaling	Meaning
A	1	Amperes
F	1	Farads
ft	0.3048	feet
H	1	Henries
Hz	1	Hertz
in	0.0254	inches

Table 4-4. Predefined Scale Factors

Scale Factor	Scaling	Meaning
meter	1	meters
meters	1	meters
metre	1	meters
metres	1	meters
mi	1609.344	miles
mil	$2.54*10^{-5}$	mils
mils	$2.54*10^{-5}$	mils
nmi	1852	nautical miles
Ohm	1	Ohms
Ohms	1	Ohms
S	1	Siemens
sec	1	seconds
V	1	Volts
W	1	Watts

Predefined Scale Factors

This type of scale factor is a predefined sequence of characters which the ADS Simulator parses as a single token. The predefined scale factors are listed in Table 4-4.

Single-character prefixes

If the first character of the scale factor is one of the legal scale-factor prefixes, the corresponding scaling is applied.The single-character prefixes are based on the metric system of scaling prefixes and are listed in Table 4-5.

For example, `3.5 GHz` is equivalent to $3.5*10^{9}$ and `12 nF` is equivalent to $1.2*10^{-8}$. Note that most of the time, the ADS Simulator ignores any characters that follow the single-character prefix. The exceptions are noted in the section on "Unrecognized Scale Factors" on page -7.

Most of these scale factors can be used without any additional characters (e.g., `3.5 G`, `12n`). This means that `m`, when used alone, stands for "milli".

The underscore _ is provided to turn off scaling. For example, `1e-9 _farad` is equivalent to 10^{-9}, and `1e-9` farad is equivalent to 10^{-24}.

Predefined scale factors are case sensitive.

Unless otherwise noted, additional characters can be appended to a predefined scale factor prefix without affecting its scaling value.

Table 4-5. Single-character prefixes

Prefix	Scaling	Meaning
T	10^{12}	tera
G	10^{9}	giga
M	10^{6}	mega
K	10^{3}	kilo
k	10^{3}	kilo
-	1	
m	10^{-3}	milli
u	10^{-6}	micro
n	10^{-9}	nano
p	10^{-12}	pico
f	10^{-15}	femto
a	10^{-18}	atto

A predefined scale factor overrides any corresponding single-character-prefix scale factor. For example, 3 mm is equivalent to $3*10^{-3}$, not $3*10^{6}$. In particular, note that M does not stand for milli, m does not stand for mega, and F does not stand for femto.

There are no scale factors for dBm, dBW, or temperature. For more information, refer to the section on “Functions” on page -17 for conversion functions.

Unrecognized Scale Factors

The ADS Simulator treats unrecognizable scale factors as equal to 1 and generates a warning message.

Scale-Factor Binding

More than one scale factor may appear in an expression, so expressions like `x in + y mil` are valid and behave properly.

Scale factors bind tightly to the preceding variable. For instance, `6 + 9 MHz` is equal to `9000006`. Use parentheses to extend the scope of a scale factor (e.g., `(6 + 9) MHz`).

Booleans

Many devices, models, and analyses have parameters that are boolean valued. Zero is used to represent false or no, whereas any number besides zero represents true or yes. The keywords **yes** and **no** can also be used.

Ground Nodes

Node 0 is assumed to be the ground node. Additional ground node aliases can be defined using the **ground** statement. Multiple **ground** statements can be used to define any number of ground aliases, but they must all occur at the top-level hierarchy in the netlist.

General Form:

Ground *[:name] node1 [... nodeN]*

Example:

```
Ground gnd
```

Global Nodes

Global nodes are user-defined nodes which exist throughout the hierarchy. The global nodes must be defined on the first lines in the netlist. They must be defined before they are used.

General Form:

globalnode *nodename1* [*nodename2*] [... *nodenameN*]

Example:

```
globalnode sumnode my_internal_node
```

Comments

Comments are introduced into an ADS Simulator file with a semicolon; they terminate at the end of the line. Any text on a line that follows a semicolon is ignored. Also, all blank lines are ignored.

Statement Order

Models can appear anywhere in the netlist. They do not have to be defined before a model instance is defined.

Some parameters expect a device instance name as the parameter value. In these cases, the device instance must already have been defined before it is referenced. If not, the device instance name can be entered as a quoted string using double quotes (").

Naming Conventions

The full name for an instance parameter is of the form:

[pathName].instanceName.parameterName[index]

where *pathName* is a hierarchical name of the form

[pathName].subcircuitInstanceName

The same naming convention is used to reference nodes, variables, expressions, functions, device terminals, and device ports.

For device terminals, the terminal name can be either the terminal name given in the device description, or *tn* where *n* is the terminal number (the first terminal in the description is terminal 1, etc.). Device ports are referenced by using the name *pm*, where *m* is the port number (the first pair of terminals in the device description is port 1, etc.).

Note that *t1* and *p1* both correspond to the current flowing into the first terminal of a device, and that *t2* corresponds to the current flowing into the second terminal. If terminals one and two define a port, then the current specified by *t2* is equal and opposite to the current specified by *t1* and *p1*.

Currents

The only currents that can be accessed for simulation, optimization, or output purposes are the state currents.

State currents

Most devices are voltage controlled, that is, their terminal currents can be calculated given their terminal voltages. Circuits that contain only voltage-controlled devices can be solved using node analysis. Some devices, however, such as voltage sources, are not voltage controlled. Since the only unknowns in node analysis are the node voltages, circuits that contain non-voltage-controlled devices cannot be solved using node analysis. Instead, modified node analysis is used. In modified node analysis, the unknown vector is enlarged. It contains not only the node voltages but the branch currents of the non-voltage-controlled devices as well. The branch currents that appear in the vector of unknowns are called state currents. Since the ADS Simulator uses modified node analysis, the values of the state currents are available for output.

If the value of a particular current is desired but the current is not a state current, insert a short in series with the desired terminal. The short does not affect the behavior of the circuit but does create a state current corresponding to the desired current.

To reference a state current, use the device instance name followed by either a terminal or port name. If the terminal or port name is not specified, the state current defaults to the first state current of the specified device. Note that this does not correspond to the current through the first port of the device whenever the current through the first port is not a state current. For some applications, the positive state current must be referenced, so a terminal name of *t1* or *t3* is acceptable but not *t2*. Using port names avoids this problem. The convention for current polarity is that positive current flows into the positive terminal.

Instance Statements

General Form:

type [:name] node1 ... nodeN [[param=value] ...]

type [:name] [[param=value] ...]

Examples:

```
ua741:OpAmp in out out
C:C1 2 3 C=10pf
HB:Distortion1 Freq=10GHz
```

The instance statement is used to define to the ADS Simulator the information unique to a particular instance of a device or an analysis. The instance statement consists of the instance type descriptor and an optional name preceded by a colon. If it is a device instance with terminals, the nodes to which the terminals of the instance are connected come next. Then the parameter fields for the instance are defined. The parameters can be in any order. The nodes, though, must appear in the same order as in the device or subcircuit definition.

The type field may contain either the ADS Simulator instance type name, or a user-supplied model or subcircuit name. The name can be any valid name, which means it must begin with a letter, can contain any number of letters and digits, must not contain any delimiters or non alphanumeric characters, and must not conflict with other names including node names.

Model Statements

General Form:

model *name type* [[*param* = *value*] ...]

Examples:

```
model NPNbjt bjt NPN=yes Bf=100 Js=0.1fa
```

Often characteristics of a particular type of element are common to a large number of instances. For example, the saturation current of a diode is a function of the process used to construct the diode and also of the area of the diode. Rather than describing the process on each diode instantiation, that description is done once in a model statement and many diode instances refer to it. The area, which may be different for each device, is included on each instance statement. Though it is possible to have several model statements for a particular type of device, each instance may only reference at most one model. Not all device types support model statements.

The name in the *model* statement becomes the type in the *instance* statement. The type field is the ADS Simulator-defined model name. Any parameter value not supplied will be set to the model’s default value.

Most models, such as the diode or bjt models, can be instantiated with an instance statement. There are exceptions. For instance, the *Substrate* model cannot be instantiated. Its name, though, can be used as a parameter value for the *Subst* parameter of certain transmission line devices.

Subcircuit Definitions

General Form:

define *subcircuitName* (*node1* ... *nodeN*)

[**parameters** *name1* = [*value1*] ... *name n* = [*value n*]]

.

.

.

elementStatements

.

.

.

end [*subcircuitName*]

Examples:

```
define DoubleTuner (top bottom left right)
parameters vel=0.95 r=1.0 l1=.25 l2=.25
    tline:tuner1 top bottom left left len=l1 vel=vel r=r
    tline:tuner2 top bottom right right len=l2 vel=2*vel r=r
end DoubleTuner
DoubleTuner:InputTuner t1 b2 3 4 l1=0.5
```

A subcircuit is a named collection of instances connected in a particular way that can be instantiated as a group any number of times by subcircuit calls. The subcircuit call is in effect and form, an instance statement. Subcircuit definitions are simply circuit macros that can be expanded anywhere in the circuit any number of times. When an instance in the input file refers to a subcircuit definition, the instances specified within the subcircuit are inserted into the circuit. Subcircuits may be nested. Thus a subcircuit definition may contain other subcircuits. However, a subcircuit definition cannot contain another subcircuit definition. All the definitions must occur at the top level.

An instance statement that instantiates a subcircuit definition is referred to as a subcircuit call. The node names (or numbers) specified in the subcircuit call are substituted, in order, for the node names given in the subcircuit definition. All instances that refer to a subcircuit definition must have the same number of nodes as are specified in the subcircuit definition and in the same order. Node names inside the subcircuit definition are strictly local unless they are a global ground defined with a **ground** statement or global nodes defined with a **globalnode** statement. A subcircuit definition with no nodes must still include the parentheses **( )**.

Parameter specification in subcircuit definitions is optional. Any parameters that are specified are referred to by name followed by an equals sign and then an optional default value. If, when making a subcircuit call in your input file, you do not specify a particular parameter, then this default value is used in that instance. Subcircuit parameters can be used in expressions within the subcircuit just as any other variable.

Subcircuits are a flexible and powerful way of developing and maintaining hierarchical circuits. Parameters can be used to modify one instance of a subcircuit from another. Names within a subcircuit can be assigned without worrying about conflicting with the same name in another subcircuit definition. The full name for a node or instance include its path name in addition to its instance name. For example, if the above subcircuit is included in `subckt2` which is itself included in `subckt1`, then the full path name of the length of the first transmission line is `subckt1.subckt2.tuner1.len`.

Only enough of the path name has to be specified to unambiguously identify the parameter. For example, an analysis inside `subckt1` can reference the length by `subckt2.tuner1.len` since the name search starts from the current level in the hierarchy. If a reference to a name cannot be resolved in the local level of hierarchy, then the parent is searched for the name, and so on until the top level is searched. In this way, a sibling can either inherit its parent's attributes or define its own.

Expression Capability

The ADS Simulator has a powerful and flexible symbolic expression capability, called *VarEqn*, which allows the user to define variables, expressions, and functions in the netlist. These can then be used to define other *VarEqn* expressions and functions, to specify device parameters and optimization goals, etc.

The names for *VarEqn* variables, expressions, and functions follow the same hierarchy rules that instance and node names do. Thus, local variables in a subcircuit definition can assume values that differ from one instance of the subcircuit to the next.

Functions and expressions can be defined either globally or locally anywhere in the hierarchy. All variables are local by default. Local variables are known in the subcircuit in which they are defined, and all lower subcircuits; they are not known at higher levels. Expressions defined at the root (the top level) are known everywhere within the circuit. To specify an expression to be global the **global** keyword must precede the expression. The **global** keyword causes the variable to be defined at the root of the hierarchy tree regardless of the lexical location.

Examples:

```
global exp1 = 2.718
```

The expression capability includes the standard math operations of + - / * ^ in addition to parenthesis grouping. Scale factors are also allowed in general expressions and have higher precedence than any of the math operators. For more information, refer to the previous section on "Units and Scale Factors" on page -5.

Constants

An integer constant is represented by a sequence of digits optionally preceded by a negative sign (e.g, `14`, `-3`).

A real number contains a decimal point and/or an exponential suffix using the `e` notation (e.g, `14.0`, `-13e-10`).

The only complex constant is the predefined constant `j` which is equal to the square root of -1. It can be used to generate complex constants from real and integer constants (e.g., `j*3`, `9.1 + j*1.2e-2`). The predefined functions `complex()` and `polar()` can also be used to enter complex constants into an expression.

A string constant is delimited by single quotes (e.g.,`'string'`,`'this is a string'`).

Predefined Constants

Table 4-6. Predefined Constants

Constant	Definition	Constant	Definition
`boltzmann`	Boltzmann's constant	`ln10`	2.30...
`c0`	Speed of light in a vacuum	`j`	Square root of -1

Table 4-6. Predefined Constants

DF_DefaultInt	Reference to default int value defined in Data Flow controller	pi	3.14...
DF_ZERO_OHMS	Symbol for use as zero ohms	planck	Planck's constant
e	2.718...	qelectron	Charge of an electron
e0	Permittivity of a vacuum	tinyReal	Smallest real number
hugeReal	Largest real number	u0	Permeability of a vacuum

Variables

General Form:

variableName = constantExpression

Examples:

```
x1 = 4.3inches + 3mils
syc_a = cos(1.0+sin(pi*3))
Zin = 7.8k - j*3.2k
```

The type of a variable is determined by the type of its value. For example, x=1 is an integer, x=1+j is complex, and x = "tuesday" is a string.

Predefined Variables

In addition to the predefined constants, there are several predefined global variables. Since they are variables, they can be modified and swept.

__fdd	Flag to indicate a new FDD instance
__fdd_v	Flag to indicate updated FDD state vars
_ac_state	Is analyses in ac state
_c1 to _c30	Symbolic controlling current
_dc_state	Is analyses in dc state
_freq1 to _freq12	Fundamental frequency
_harm	Harmonic number index for sources and FDD
_hb_state	Is analyses in harmonic balance state
_p2dInputPower	Port input power for P2D simulation
_sigproc_state	Is analyses in signal processing state

_sm_state	Is analyses in sm state	
_sp_state	Is analyses in sparameter analysis state	
_tr_state	Is analyses in transient state	
CostIndex	Index for optimization cost plots	
DF_Value	Reference to corresponding value defined in Data Flow controller	
DefaultValue	Signal processing default parameter value	
DeviceIndex	Device Index used for noise contribution or DC OP output	
dcSourceLevel	used for DC source-level sweeping	
doeindex	Index for Design of Experiment sweeps	
freq	The frequency in Hertz of the present simulation	(1MHz)
logNodesetScale	Used for DC nodeset simulation	
logRshunt	Used for DC Rshunt sweeping	
mcTrial	Trial counter for Monte Carlo based simulations	
noisefreq	The spectral noise analysis frequency	
Nsample	Signal processing analysis sample number	
optIter	Optimization job iteration counter	
temp	The ambient temperature, in degrees Celsius.	(25^{o} C)
time	The analysis time	
timestep	The analysis time step	
tranorder	The transient analysis integration order	
ScheduleCycle	Signal processing schedule cycle number	
sourcelevel	The relative attenuation of the spectral sources	(1.0)
ssfreq	The small-signal mixer analysis frequency	
_v1 to _v19	State variable voltages used by the sdd device	
_i1 to _i19	State variable currents used by the sdd device	
mc_index	Index variable used by Monte Carlo controller	

The `sourcelevel` variable is used by the spectral analysis when it needs to gradually increase source power from 0 to full scale to obtain convergence. It can be used by the user to sweep the level of ALL spectral source components, but is not recommended. The `_v` and `_i` variables should only be used in the context of the sdd device.

Expressions

General Form:

expressionName = *nonconstantExpression*

Examples:

```
x1 = 4.3 + freq;
syc_a = cos(1.0+sin(pi*3 + 2.0*x1))
Zin = 7.8 ohm + j*freq * 1.9 ph
y = if (x equals 0) then 1.0e100 else 1/x endif
```

The main difference between expressions and variables is that a variable can be directly swept and modified by an analysis but an expression cannot. Note however, that any instance parameter that depends on an expression is updated whenever one of the variables that the expression depends upon is changed (e.g., by a sweep).

Predefined Expressions

`gaussian =`	`_gaussian_tol(10.0)`	default gaussian distribution
`nfmin =`	`_nfmin()`	the minimum noise figure
`omega =`	`2.0*pi*freq`	the analysis frequency
`rn =`	`_rn()`	the noise resistance
`sopt =`	`_sopt`	the optimum noise match
`tempkelvin =`	`temp + 273.15`	the analysis temperature
`uniform =`	`_uniform_tol(10.0)`	default uniform distribution

Functions

General Form:

functionName(*[arg1, ..., argn]*) = *expression*

Examples:

```
y_srl(freq, r, l) = 1.0/(r + j*freq*l)
expl(a,b) = exp(a)*step(b-a) + exp(b)*(a-b-1)*step(a-b)
```

In *expression*, the function's arguments can be used, as can any other *VarEqn* variables, expressions, or functions.

Predefined Functions

_discrete_density(...)	user-defined discrete density function
_gaussian([*mean*, *sigma*, *lower_n_sigmas*, *upper_n_sigmas*, *lower_n_sigmas_del*, *upper_n_sigmas_del*])	gaussian density function
_gaussian_tol[*percent_tol*, *lower_n_sigmas*, *upper_n_sigmas*, *lower_percent_tol*, *upper_percent_tol*, *lower_n_sigmas_del*, *upper_n_sigmas_del*])	gaussian density function (tolerance version)
_get_fnom_freq(...)	Get analysis frequency for FDD carrier frequency index and harmonic
_lfsr(*x*, *y*, *z*)	linear feedback shift register (trigger, seed, taps)
_mvgaussian(...)	multivariate gaussian density function (correlation version)
_mvgaussian_cov(...)	multivariate gaussian density function (covariance version)
_n_state(*x*, *y*)	_n_state(*arr*, *val*) array index nearest value
_pwl_density(...)	user-defined piecewise-linear density function
_pwl_distribution(...)	user-defined piecewise-linear distribution function
_randvar(*distribution*, *mcindex*, [*nominal*, *tol_percent*, *x_min*, *x_max*, *lower_tol*, *upper_tol*, *delta_tol*, *tol_factor*])	random variable function
_shift_reg(*x*, *y*, *z*, *t*)	(trigger, mode(ParIn:MSB1st), length, input)
_uniform([*lower_bound*, *upper_bound*])	uniform density function
_uniform_tol([*percent_tol*, *lower_tol*, *upper_tol*])	uniform density function (tolerance version)
abs(*x*)	absolute value function

access_all_data(...)	datafile indep+dep lookup/interpolation function
access_data(...)	datafile dependents' lookup/interpolation function
arcsinh(*x*)	arcsinh function
arctan(*x*)	arctan function
atan2(*y*, *x*)	arctangent function (two real arguments)
awg_dia(*x*)	wire gauge to diameter in meters
bin(*x*)	function convert a binary to integer
bitseq(*time*, [*clockfreq*, *trise*, *tfall*, *vlow*, *vhigh*, *bitseq*])	bitsequence function
complex(*x*, *y*)	real-to-complex conversion function
conj(*x*)	complex-conjugate function
cos(*x*)	cosine function
cos_pulse(*time*, [*low*, *high*, *delay*, *rise*, *fall*, *width*, *period*])	periodic cosine shaped pulse function
cosh(*x*)	hyperbolic cosine function
cot(*x*)	cotangent function
coth(*x*)	hyperbolic cotangent function
ctof(*x*)	convert Celsius to Fahrenheit
ctok(*x*)	convert Celsius to Kelvin
cxform(*x*, *y*, *z*)	transform complex data
damped_sin(*time*, [*offset*, *amplitude*, *freq*, *delay*, *damping*, *phase*])	damped sin function
db(*x*)	decibel function
dbm(*x*, *y*)	convert voltage and impedance into dbm
dbmtoa(*x*, *y*)	convert dbm and impedance into short circuit current
dbmtov(*x*, *y*)	convert dbm and impedance into open circuit voltage
dbmtow(*x*)	convert dBm to Watts
dbpolar(*x*, *y*)	(dB,angle)-to-rectangular conversion function

dbwtow(*x*)	convert dBW to Watts
deembed(*x*)	deembedding function
deg(*x*)	radian-to-degree conversion function
dep_data(*x*, *y*, [*z*])	dependent variable value
dphase(*x*, *y*)	Continuous phase difference (radians) between x and y
dsexpr(*x*, *y*)	Evaluate a dataset expression to an hpvar
dstoarray(*x*, [*y*])	Convert an hpvar to an array
echo(*x*)	echo-arguments function
erf_pulse(*time*, [*low*, *high*, *delay*, *rise*, *fall*, *width*, *period*])	periodic error function shaped pulse function
eval_poly(*x*, *y*, *z*)	polynomial evaluation function
exp(*x*)	exponential function
exp_pulse(*time*, [*low*, *high*, *delay1*, *tau1*, *delay2*, *tau2*])	exponential pulse function
fread(*x*)	raw-file reading function
ftoc(*x*)	convert Fahrenheit to Celsius
ftok(*x*)	convert Fahrenheit to Kelvin
get_array_size(*x*)	Get the size of the array
get_attribute(...)	value of attribute of a set of data
get_block(*x*, *y*)	HPvar tree from block name function
get_fund_freq(*x*)	Get the frequency associated with a specified fundamental index
get_max_points(*x*, *y*)	maximum points of independent variable
imag(*x*)	imaginary-part function
index(*x*, *y*, [*z*, *t*])	get index of name in array
innerprod(...)	inner-product function
int(*x*)	convert-to-integer function
itob(*x*, [*y*])	convert integer to binary
jn(*x*, *y*)	bessel function

ktoc(*x*)	convert Kelvin to Celsius
ktof(*x*)	convert Kelvin to Fahrenheit
length(*x*)	returns number of elements in array
limit_warn([*x*, *y*, *z*, *t*, *u*])	limit, default and warn function
list(...)	
ln(*x*)	natural log function
log(*x*)	log base 10 function
mag(*x*)	magnitude function
makearray(...)	(1:real-2:complex-3:string, y, z..) or (array, startIndex, stopIndex)
max(*x*, *y*)	maximum function
min(*x*, *y*)	minimum function
multi_freq(*time*, *amplitude*, *freq1*, *freq2*, *n*, [*seed*])	multifrequency function
names(*x*, *y*)	array of names of indepVars and/or depVars in dataset
norm(*x*)	norm function
phase(*x*)	phase (in degrees) function
phase_noise_pwl(...)	piecewise-linear function for computing phase noise
phasedeg(*x*)	phase (in degrees) function
phaserad(*x*)	phase (in radians) function
polar(*x*, *y*)	polar-to-rectangular conversion function
polarcpx(...)	polar to rectangular conversion function
pulse(*time*, [*low*, *high*, *delay*, *rise*, *fall*, *width*, period])	periodic pulse function
pwl(...)	piecewise-linear function
pwlr(...)	piecewise-linear-repeated function
rad(*x*)	degree-to-radian conversion function
ramp(*x*)	ramp function
read_data(...)	read_data("file-dataset", "locName", "fileType")
read_lib(...)	read_lib("libName", "item", "fileType")

real(*x*)	real-part function
rect(*x*, *y*, *z*)	rectangular pulse function
rem(...)	remainder function
ripple(*x*, *y*, *z*, *v*)	ripple(amplitude, intercept, period, variable) sinusoidal ripple function
rms(...)	root-mean-square function
rpsmooth(*x*)	rectangular-to-polar smoothing function
scalearray(*x*, *y*)	scalar times a vector (array) function
setDT(*x*)	Turns on discrete time transient mode (returns argument)
sffm(*time*, [*offset*, *amplitude*, *carrier_freq*, *mod_index*, *signal_freq*])	signal frequency FM
sgn(*x*)	signum function
sin(*x*)	sine function
sinc(*x*)	sin(x)/x function
sinh(*x*)	hyperbolic sine function
sprintf(*x*, *y*)	formatted print utility
sqrt(*x*)	square root function
step(*x*)	step function
tan(*x*)	tangent function
tanh(*x*)	hyperbolic tangent function
vswrpolar(*x*, *y*)	(VSWR,angle)-to-rectangular conversion function

Note The *VarEqn* trigonometric functions always expect the argument to be specified in radians. If the user wants to specify the angle in degrees then the *VarEqn* function `deg()` can be used to convert radians to degrees or the *VarEqn* function `rad()` can be used to convert degrees to radians.

Detailed Descriptions of the Predefined Functions

`_discrete_density` (x_1, p_1, x_2, p_2, …) allows the user to define a discrete density distribution: returns x_1 with probability p_1, x_2 with probability p_2, etc. The x_n, p_n pairs needn't be sorted. The p_ns will be normalized automatically.

`_gaussian( [`*mean*, *sigma*, *lower_n_sigmas*, *upper_n_sigmas*, *lower_n_sigmas_del*, *upper_n_sigmas_del*`] )` returns a value randomly distributed according to the standard bell-shaped curve. *mean* defaults to 0. *sigma* defaults to 1. *lower_n_sigmas*, *upper_n_sigmas* define truncation limits (default to 3). *lower_n_sigmas_del* and *upper_n_sigmas_del* define a range in which the probability is zero (a bimodal distribution). `_gaussian_tol( [`*percent_tol*, *lower_n_sigmas*, *upper_n_sigmas*, *lower_percent_tol*, *upper_percent_tol*, *lower_n_sigmas_del*, *upper_n_sigmas_del*`] )` is similar, but *percent_tol* defines the percentage tolerance about the nominal value (which comes from the RANDVAR expression).

`_get_fnom_freq(`*x*`)` returns the actual analysis frequency associated with the carrier frequency specified in the surrounding FDD context. If *x* is negative, it is the carrier frequency index. If *x* is positive, it is the harmonic index.

`_mvgaussian`(*N*, $mean_1$, … $mean_N$, $sigma_1$, … $sigma_N$, $correlation_{1,2}$, …, $correlation_{1,N}$, …, $correlation_{N-1,N}$) multivariate gaussian density function (correlation version). Returns an *N* dimensional vector. The correlation coefficient matrix must be positive definite. _mvgaussian_cov(*N*, $mean_1$, … $mean_N$, $sigma_1$, … $sigma_N$, $covariance_{1,2}$, ..., $covariance_{1,N}$, ..., $covariance_{N-1,N}$) is similar, but defined in terms of covariance. The covariance matrix must be positive definite.

_pwl_density(x_1, p_1, x_2, p_2, …) returns a value randomly distributed according to the piecewise-linear density function with values p_n at x_n, i.e. it will return x_n with probability p_n and return

$$x_n + \varepsilon \text{ with probability } p_n + \varepsilon\frac{p_{n+1} - p_n}{x_{n+1} - x_n}$$

The x_n, p_n pairs needn't be sorted. The p_ns will be normalized automatically. _pwl_distribution(x_1, p_1, x_2, p_2, …) is similar, but is defined in terms of the distribution values. It will return a value less than or equal to x_n with probability p_n. The x_n, p_n pairs will be sorted in increasing x_n order. After sorting, the p_ns should never decrease. The p_ns will be normalized so that p_N=1.

_randvar(*distribution*, *mcindex*, [*nominal*, *tol_percent*, *x_min*, *x_max*, *lower_tol*, *upper_tol*, *delta_tol*, *tol_factor*]) returns a value randomly distributed according to the *distribution*. The value will be the same for a given value of *mcindex*. The other parameters are interpreted according to the *distribution*.

_shift_reg(*x*, *y*, *z*, *t*) implements a *z*-bit shift register. *x* specifies the trigger. *y* = 0 means LSB First, Serial To Parallel, 1 means MSB First, Serial To Parallel, 2 means LSB First, Parallel to Serial, 3 means MSB First, Parallel to Serial. *t* is the input (output) value.

_uniform([*lower_bound*, *upper_bound*]) returns a value between *lower_bound* and *upper_bound*. All such values are equally probable. _uniform_tol([*percent_tol*, *lower_tol*, *upper_tol*]) is similar, but tolerance version.

access_all_data(*InterpMode*, *source*, $indep_1$, dep_1 ...) datafile independent and dependent lookup/interpolation function.

access_data(*InterpMode*, *nData*, *source*, dep_1 ...) datafile dependents' lookup/interpolation function.

bin(*String*) calculates the integer value of a sequence of 1's and 0's. For example bin('11001100') = 204. The argument of the bin function must be a string denoted by single quotes. The main use of the bin function is with the *System Model Library* to define an integer which corresponds to a digital word.

cxform(*x*, *OutFormat*, *InFormat*) transform complex data *x* from format *InFormat* to format *OutFormat*. The values for *OutFormat* and *InFormat* are 0: real and imaginary, 1: magnitude (linear) and phase (degrees), 2: magnitude (linear) and phase (radians), 3: magnitude (dB) and phase (degrees), 4: magnitude (dB) and phase (radians), 5: magnitude (SWR) and phase (degrees), 6: magnitude (SWR) and phase (radians). For example, to convert linear magnitude and phase in degrees to real and imaginary parts:

```
result = cxform(invar, 0, 1)
```

damped_sin(*time*, [*offset*, *amplitude*, *freq*, *delay*, *damping*, *phase*]). Refer to "Transient Source Functions" on page -28.

The function db(*x*) is a shorthand form for the expression: 20log(mag(*x*)).

The deembed(*x*) function takes an array, *x*, of four complex numbers (the 2-port S-parameter array returned from the *VarEqn* interp() function) and returns an array of equivalent de-embedding S-parameters for that network. The array must be of length four (2 x 2--two-port data only), or an error message will result. The transformation used is:

$$S_{11}^{-1} = \frac{S_{11}}{det}$$

$$S_{21}^{-1} = \frac{S_{21}}{det}$$

$$S_{12}^{-1} = \frac{S_{12}}{det}$$

$$S_{22}^{-1} = \frac{S_{22}}{det}$$

where *det* is the determinant of the 2 x 2 array.

WARNING: This transformation assumes that the S-parameters are derived from equal port termination impedances. **This transformation does not work when the port impedances are unequal.**

The function deg(*x*) converts from radians to degrees.

dphase(*x*, *y*) Calculates phase difference phase(*x*)-phase(*y*) (in radians).

dsexpr(*x*, *y*) Evaluate *x*, a DDS expression, to an hpvar. *y* is the default location data directory.

echo(*x*) prints argument on terminal and returns it as a value.

erf_pulse(*time*, [*low*, *high*, *delay*, *rise*, *fall*, *width*, *period*]) periodic pulse function, edges are error function (integral of Gaussian) shaped.

eval_poly(*x*, *y*, *z*) y is a real number. z is an integer that describes what to evaluate: -1 means the integral of the polynomial, 0 means the polynomial itself, +1 means the derivative of the polynomial. x is a *VarEqn* array that contains real numbers. The polynomial is $x_0 + x_1 y + x_2 y^2 + x_3 y^3 \ldots$

exp_pulse(*time*, [*low*, *high*, *delay1*, *tau1*, *delay2*, *tau2*]) Refer to “Transient Source Functions” on page -28.

get_fund_freq(*fund*) returns the value of frequency (in Hertz) of a given fundamental defined by *fund*.

index(*nameArray*, "varName", [*caseSense*, *length*]) returns position of "varName" in *nameArray*, -1 if not found. *caseSense* sets case-sensitivity, defaults to yes. *length* sets how many characters to check, defaults to 0 (all).

innerprod(x, y) forms the inner product of the vectors x and y:

$$innerprod(x, y) = \sum_{i=0}^{n} x_i^* y_i$$

j and k are optional integers which specify a range of harmonics to include in the calculation:

$$innerprod(x, y, j, k) = \sum_{i=j}^{k} x_i^* y_i$$

j defaults to 0 and k defaults to infinity.

int(x) Truncates the fractional part of x.

itob(x, [*bits*]) convert integer x to *bits-bit* binary string.

The function jn(n, x) is the n-th order bessel function evaluated at x.

limit_warn([*Value*, *Min*, *Max*, *default*, *Name*]) sets *Value* to *default*, if not set. Limits it to *Min* and *Max* and generates a warning if the value is limited.

makearray(*arg1*[,*arg2*,..] creates an array with elements defined by *arg1* to *argN* where N can be any number of arguments. The data type of *args* must be Integer, Real, or Complex and the same for all *args*.

```
word = bin('1101')
fibo = makearray(0,1,1,2,3,5,8,word)
foo = fibo[0]
```

multi_freq(*time*, *amplitude*, *freq1*, *freq2*, *n*, [*seed*]) *seed* defaults to 1. If it is 0, phase is set to 0, otherwise it is used as a seed for a randomly-generated phase.

norm(x) returns the L-2 norm of the spectrum x:

$$norm(x) = \sqrt{innerprod(x, x)}$$

j and k are optional integers which specify a range of harmonics to include in the calculation:

$$norm(x, j, k) = \sqrt{innerprod(x, x, j, k)}$$

j defaults to 0 and k defaults to infinity.

phase(*x*) is the same as phasedeg(*x*).

The function phasedeg(*x*) returns phase in degrees.

The function phaserad(*x*) returns phase in radians.

The function polarcpx(*x*[,*leave_as_real*]) takes a complex argument, assumes that the real and complex part of the argument represents *mag* and *phase* (in radians) information, and converts it to real/imaginary. If the argument is real or integer instead of complex, the imaginary part is assumed to be zero. However, if the optional *leave_as_real* variable is specified, and is the value "1" (note that the legal values are "0" and "1" only), a real argument will be not be converted to a complex one.

pulse(*time*, [*low*, *high*, *delay*, *rise*, *fall*, *width*, *period*]) Refer to "Transient Source Functions" on page -28.

pwl(...) piecewise-linear function. Refer to "Transient Source Functions" on page -28.

pwlr(...) piecewise-linear-repeated function.

The function rect(*t*, *tc*, *tp*) is pulse function of variable *t* centered at time *tc* with duration *tp*.

The function rad(*x*) converts from degrees to radians.

ramp(*x*) 0 for $x < 0$, x for $x \geq 0$

read_data(*source*, *locName*, [*fileType*]) returns data from a file or dataset. *source* = "file" --- "dataset". *locName* is the name of the source. *fileType* specifies the file type.

read_lib(*libName*, *locName*, [*fileType*]) returns data from a library. *libName* is the name of the library. *locName* is the name of the source. *fileType* specifies the file type. read_lib("libName", "item", "fileType")

rect(*x*, *y*, *z*) Returns:

Table 4-7.

z	|*x* - *y*| < |*z*|	|*x* - *y*| > |*z*|
> 0	1	0
< 0	0	1

rem(*x*, [*y*]) Returns remainder of dividing *x*/*y*. *y* defaults to 0 (which returns *x*).

rms(*x*) returns the RMS value (including DC) of the spectrum *x*:

$$rms(x) = \frac{norm(x)}{\sqrt{2.0}}$$

j and *k* are optional integers which specify a range of harmonics to include in the calculation:

$$rms(x, j, k) = \frac{norm(x, j, k)}{\sqrt{2.0}}$$

j defaults to 0 and *k* defaults to infinity.

The function rpsmooth(*x*) takes a *VarEqn* pointer (one returned by readraw()), converts to polar format the rectangular data given by the *VarEqn* pointer, and smooths out ‘phase discontinuities’.

WARNING: This function uses an algorithm that assumes that the first point is correct (i.e., not off by some multiple of 2π) and that the change in phase between any two adjacent points is less than π. This interpolation will not work well with noisy data or with data within roundoff error of zero. It should be used only with S-parameters in preparation for interpolation or extrapolation by one of the interpolation functions like interp1(). Also note that the result is left in a polar ‘mag/phase’ format stored in a complex number; the real part is magnitude, and the imaginary part is phase. The polarcpx() function must be used to convert the result of the rpsmooth() function back into a real/imaginary format.

---

sffm(*time*, [*offset*, *amplitude*, *carrier_freq*, *mod_index*, *signal_freq*]) Refer to “Transient Source Functions” on page -28.

The `sprintf()` function is similar to the `C` function which takes a format string for argument *s* and a print argument *x* (*x* must be a string, an integer, or a real number) and returns a formatted string. This string then may be written to the console using the `system` function with an `echo` command.

Transient Source Functions

There are several built-in functions that mimic Spice transient sources. They are:

Table 4-8.

SPICE source	ADS Simulator function
exponential	exp_pulse(*time*, *low*, *high*, *tdelay1*, *tau1*, *tdelay2*, *tau2*)

Table 4-8.

single-frequency FM	sffm(*time*, *offset*, *amplitude*, *carrier_freq*, *mod_index*, *signal_freq*)
damped sine	damped_sin(*time*, *offset*, *amplitude*, *freq*, *delay*, *damping*)
pulse	pulse(*time*, *low*, *high*, *delay*, *rise*, *fall*, *width*, *period*)
piecewise linear	pwl(*time*, *t1*, *x1*, ..., *tn*, *xn*)

There functions are typically used with the *vt* parameter of the voltage source and the *it* parameter of the current source.

exp_pulse

Examples:

```
ivs:vin n1 0 vt=exp_pulse(time)
ics:iin n1 0 it=exp_pulse(time, -0.5mA, 0.5mA, 10ns, 5ns, 20ns, 8ns)
```

Table 4-9.

Arguments for exp_pulse		
Name	Optional	Default
TIME	NO	
LOW	YES	0
HIGH	YES	1
TDELAY1	YES	0
TAU1	YES	TSTEP
TDELAY2	YES	TDELAY1 + TSTEP
TAU2	YES	TSTEP

TSTEP is the output step-time time specified on the TRAN analysis.

sffm

Examples:

```
ivs:vin n1 0 vt=sffm(time, , , , 0.5)
ics:iin n1 0 it=sffm(time, 0, 2, 1GHz, 1.2, 99MHz)
```

Table 4-10.

Arguments for sffm		
Name	Optional	Default
TIME	NO	
OFFSET	YES	0
AMPLITUDE	YES	1
CARRIER_FREQ	YES	1/TSTOP
MOD_INDEX	YES	0
SIGNAL_FREQ	YES	1/TSTOP

TSTOP is the stop time specified on the TRAN analysis.

damped_sin

Examples:

```
ivs:vin n1 0 vt=damped_sin(time)
ics:iin n1 0 it=damped_sin(time, 0, 5V, 500MHz, 50ns, 200ns)
```

Table 4-11.

Arguments for damped_sin		
Name	Optional	Default
TIME	NO	
OFFSET	YES	0
AMPLITUDE	YES	1
FREQ	YES	1/TSTOP
DELAY	YES	0
DAMPING	YES	1/TSTOP

TSTOP is the stop time specified on the TRAN analysis.

pulse

Examples:

```
ivs:vin n1 0 vt=pulse(time)
ics:iin n1 0 it=pulse(time, -5V, 5V, 500MHz, 50ns, 200ns)
```

Table 4-12.

Arguments for pulse		
Name	Optional	Default
TIME	NO	
LOW	YES	0
HIGH	YES	1
DELAY	YES	0
RISE	YES	TSTEP
FALL	YES	TSTEP
WIDTH	YES	TSTOP
PERIOD	YES	TSTOP

TSTEP is the output step-time time specified on the TRAN analysis. TSTOP is the stop time specified on the TRAN analysis.

pwl

Examples:

```
ivs:vin n1 0 vt=pulse(time, 0, 0, 1ns, 1, 10ns, 1, 15ns, 0)
ics:iin n1 0 it=pwl(time, 0, 0, 1ns, 1, 5ns, 1, 5ns, 0.5, 10ns,0.5, 15ns,
0)
```

Table 4-13.

Arguments for pwl		
Name	Optional	Default
TIME	NO	
T1	NO	
X1	NO	
T2	YES	NONE
X2	YES	NONE
. . .	. . .	. . .

Table 4-13.

TN	YES	NONE
XN	YES	NONE

Conditional Expressions

The ADS Simulator supports simple in-line conditional expressions:

`if` *boolExpr* `then` *expr* `else` *expr* `endif`
`if` *boolExpr* `then` *expr* `elseif` *boolExpr* `then` *expr* `else` *expr* `endif`

boolExpr is a boolean expression, that is, an expression that evaluates to TRUE or FALSE.

expr is any non-boolean expression.

The *else* is required (because the conditional expression must always evaluate to some value).

There can be any number of occurrences of `elseif` *expr* `then` *expr*.

A conditional expression can legally occur as the right-hand side of an expression or function definition or, if parenthesized, anywhere in an expression that a variable can occur.

Boolean operators

equals	logical equals
=	logical equals
==	logical equals
notequals	logical not equals
!=	logical not equals
not	logical negative
!	logical negative
and	logical and
&&	logical and
or	logical or
||	logical or
<	less than

>	greater than
<=	less than or equals
>=	greater than or equals

Boolean expressions

A boolean expression must evaluate to TRUE or FALSE and, therefore, must contain a relational operator (`equals, =, ==, notequals, !=, <, >, <=, or >=`).

The only legal place for a boolean expression is directly after an `if` or an `elseif`.

A boolean expression cannot stand alone, that is,

x = a > b

is illegal.

Precedence

Tightest binding: `equals, =, ==, notequals, !=, >, <, >=, <=`

`NOT, !`

`AND`

Loosest binding: `OR, ||`

All arithmetic operators have tighter binding than the boolean operators.

Evaluation

Boolean expressions are short-circuit evaluated. For example, if when evaluating *a* `and` *b*, expression *a* evaluates to FALSE, expression *b* will not be evaluated.

During evaluation of boolean expressions with arithmetic operands, the operand with the lower type is promoted to the type of the other operand. For example, in `3 equals x +j*b, 3` is promoted to complex.

A complex number cannot be used with <, >, <=, or >=. Nor can an array (and remember that strings are arrays). This will cause an evaluation-time error.

Pointers can be compared only with pointers.

Examples:

Protect against divide by zero:

```
f(a) = if a equals 0 then 1.0e100 else 1.0/a endif
```

Nested if’s #1:

```
f(mode) = if mode equals 0 then 1-a else f2(mode) endif
f2(mode) = if mode equals 1 then log(1-a) else f3(mode) endif
f3(mode) = if mode equals 2 then exp(1-a) else 0.0 endif
```

Nested if’s #2:

```
f(mode) = if mode equals 0 then 1-a elseif mode equals 1 then log(1-a)  \
elseif mode equals 2 then exp(1-a) else 0.0 endif
```

Soft exponential:

```
exp_max = 1.0e16
x_max = ln(exp_max)
exp_soft(x) = if x<x_max then exp(x) else (x+1-x_max)*exp_max endif
```

VarEqn Data Types

The four basic data types that *VarEqn* supports are integer, real, complex, and string. There is a fifth data type, pointer, that is also supported. Pointers are not allowed in an algebraic expression, except as an argument to a function that is expecting a pointer. Strings are not allowed in algebraic expressions either except that addition of strings is equivalent to catenation of the strings. String catenation is not commutative, and since *VarEqn's* simplification routines can internally change the order of operands of commutative operators, this feature should be used cautiously. It will most likely be replaced by an explicit catenation function.

Type conversion

The data type of a *VarEqn* expression is determined at the time the expression is evaluated and depends on the data types of the terms in the expression. For example, let y=3*x^2. If x is an integer, then y is integer-valued. If x is real, then y is real-valued. If x is complex, then y is complex-valued.

As another example, let `y=sqrt(2.5*x)`. If `x` is a positive integer, then `y` evaluates to a real number. If, however, `x` is a negative integer, then `y` evaluates to a complex number.

There are some special cases of type conversion:

- If either operand of a division is integer-valued, it is promoted to a real before the division occurs. Thus, `2/3` evaluates to `0.6666....`

- The built-in trigonometric, hyperbolic, and logarithmic functions never return an integer, only a real or complex number.

“C-Preprocessor”

Before being interpreted by the ADS Simulator, all input files are run through a built-in preprocessor based upon a C preprocessor. This brings several useful features to the ADS Simulator, such as the ability to define macro constants and functions, to include the contents of another file, and to conditionally remove statements from the input. All C preprocessor statements begin with # as the first character.

Unfortunately, for reasons of backward compatibility, there is no way to specify include directories. The standard C preprocessor “`-I`” option is not supported; instead, “`-I`” is used to specify a file for inclusion into the netlist.

File Inclusion

Any source line of the form

#include "*filename*"

is replaced by the contents of the file *filename*. The file must be specified with an absolute path or must reside in either the current working directory or in `/$HPEESOF_DIR/circuit/components/`.

Library Inclusion

The C preprocessor automatically includes a library file if the `-N` command line option is not specified and if such a file exists. The first file found in the following list is included as the library:

```
$HPEESOF_DIR/circuit/components/gemlib
$EESOF_DIR/circuit/components/gemlib
$GEMLIB
.gemlib
~/.gemlib
~/gemini/gemlib
```

A library file is specified by the user using the `-I`*filename* command line option. More than one library may be specified. Specifying a library file prevents the ADS Simulator from including any of the above library files.

Macro Definitions

A macro definition has the form;

`#define` *name replacement-text*

It defines a macro substitution of the simplest kind--subsequent occurrences of the token *name* are replaced by *replacement-text*. The name consists of alphanumeric characters and underscores, but must not begin with a numeric character; the replacement text is arbitrary. Normally the replacement text is the rest of the line, but a long definition may be continued by placing a “\” at the end of each line to be continued. Substitutions do not occur within quoted strings. Names may be undefined with

`#undef` *name*

It is also possible to define macros with parameters. For example,

```
#define to_celcius(t) (((t)-32)/1.8)
```

is a macro with the formal parameter `t` that is replaced with the corresponding actual parameters when invoked. Thus the line

```
options temp=to_celcius(77)
```

is replaced by the line

```
options temp=(((77)-32)/1.8)
```

Macro functions may have more than one parameter, but the number of formal and actual parameters must match.

Macros may also be defined using the `-D` command line option.

Conditional Inclusion

It is possible to conditionally discard portions of the source file. The `#if` line evaluates a constant integer expression, and if the expression is non-zero, subsequent lines are retained until an `#else` or `#endif` line is found. If an `#else` line is found, any lines between it and the corresponding `#endif` are discarded. If the expression evaluates to zero, lines between the `#if` and `#else` are discarded, while those between the `#else` and `#endif` are retained. The conditional inclusion statements nest to an arbitrary level of hierarchy. The following operators and functions can be used in the constant expression;

!	Logical negation.
||	Logical or.
&&	Logical and.
==	Equal to.
!=	Not equal to.
>	Greater than.
<	Less than.
>=	Greater than or equal to.
<=	Less than or equal to.
+	Addition.
defined(x)	1 if x defined, 0 otherwise.

The `#ifdef` and `#ifndef` lines are specialized forms of `#if` that test whether a name is defined.

WARNING: Execution of preprocessor instructions depend on the order in which they appear on the netlist. When using preprocessor statements make sure that they are in the proper order. For example, if an #ifdef statement is used to conditionally include part of a netlist, the corresponding #define statement is contained in a separate file and #include is used to include the content of the file into the netlist, the #include statement will have to appear before the #ifdef statement for the expression to evaluate correctly.

Data Access Component

The Data Access Component provides a clean, unified way to access tabular data from within a simulation. The data may reside in either a text file of a supported, documented format (e.g. discrete MDIF, model MDIF, Touchstone, CITIfile), or a dataset. It provides a variety of access methods, including lookup by index/value, as well as linear, cubic spline and cubic interpolation modes, with support for derivatives.

The Data Access Component provides a "handle" with which one may access data from either a text file or dataset for use in a simulation. The DAC is implemented as a cktlib subcircuit fragment with internally known expressions names (e.g. _DAC, _TREE) that are assigned via *VarEqn* calls such as `read_data()` and `access_all_data()`. The accessed data can be used by other components (including models, devices, variables, subcircuit calls and other DAC instances) in the netlist, either by the specific file syntax or via the *VarEqn* function `dep_data()`.

The DAC can also be used to supply parameters to device and model components from text files and datasets. In this case, the `AllParams` device/model parameter is used to refer to a DAC component. The component's parameters will then be accessed from the DAC and supplied to the instance. Care is taken to ensure that only matching (between parameter names in the component definition and DAC dependent column names) data is used. Also, parameter data can be assigned "inline" - as is usually done - in which case the inline data takes precedence over the DAC data.

As the DAC component is composed of just a parameterized subcircuit, it allows alterations (sweep, tune, optimize, yield) of its parameters. Consequently any component that uses DAC data via file, `dep_data()` or `AllParams` will automatically be updated when a DAC parameter is altered. A caveat with sweeping over files using `AllParams` is that all the files must contain the same number of dependent columns of data.

Below is an example definition of a simple DAC component that accesses discrete values from a text file:

```
#uselib "ckt" , "DAC"
DAC:DAC1  File="C:\jeffm\ADS_testing\ADS13_test_prj/.\data\SweptData.ds"
Type="dataset" Block="S" InterpMode="linear" InterpDom="ri" iVar1="X"
iVal1=X iVar2="freq" iVal2=freq
S_Port:S2P1  _net1 0 _net6 0 S[1,1]=file{DAC1, "S[1,1]"}
S[1,2]=file{DAC1,"S[1,2]"} S[2,1]=1 S[2,2]=0 Recip=no


dindex = 1
DAC:atc1 File="vdcr.mdf" Type="dscr" \
InterpMode="index_lookup" iVar1=1 iVal1=dindex
```

And its use to provide the resistance value to a pair of circuit components:

```
R:R1 n1 0 R=file{atc1, "R"} kOhm
R:R2 n1 0 R=dep_data(atc1, "R") kOhm
Here, it provides the value to a variable:
V1 = file{atc1, "Vdc"}
```

V1 could be used elsewhere in the circuit, as expected.

In this example, a scaling factor applied to the result of a DAC access is shown:

```
File = "atc.mdf"
Type = "dscr"
Mode="index_lookup"
Cnom = "Cnom"
DAC:atc_s File=File Type=Type InterpMode=Mode iVar1=1 iVal1 = Cs_row
C:Cs n1 n2 C=file{atc_s, Cnom} Pf
```

In this example, a use of `AllParams` is shown to enter model parameters from a text file:

```
File = "c:\gemini\vdcr.mdf"
Type = "dscr"
Mode="index_lookup"
DAC:dac1 File=File Type=Type InterpMode=Mode iVar1=1 iVal1 = ix
model rm1 R_Model R=0 AllParams = dac1._DAC
rm1:rm1i1 n3 0
```

Reserved Words

The words on the following pages have built-in meaning and should not be defined or used in a way not consistent with their pre-defined meaning. They are listed in alphabetical order in Table 4-14 for convenience.

Table 4-14. ADS Reserved Words

"A" on page E-41	"B" on page E-41	"C" on page E-41	"D" on page E-42	"E" on page E-42	"F" on page E-42	"G" on page E-43	"H" on page E-43	"I" on page E-43	"J" on page E-43
"K" on page E-43	"L" on page E-43	"M" on page E-43	"N" on page E-45	"O" on page E-46	"P" on page E-46	"Q" on page E-47	"R" on page E-47	"S" on page E-47	"T" on page E-49
"U" on page E-49	"V" on page E-50	"W" on page E-50	"X" on page E-50	"Y" on page E-50	"Z" on page E-50	"a" on page E-54	"b" on page E-54	"c" on page E-54	"d" on page E-55
"e" on page E-55	"f" on page E-56	"g" on page E-56	"h" on page E-56	"i" on page E-56	"j" on page E-57	"k" on page E-57	"l" on page E-57	"m" on page E-58	"n" on page E-58

Table 4-14. ADS Reserved Words

“o” on page E-58	“p” on page E-58	“q” on page E-59	“r” on page E-59	“s” on page E-59	“t” on page E-60	“u” on page E-60	“v” on page E-60	“w” on page E-61	“x” on page E-61
“y” on page E-61	“z” on page E-61	“__” on page E-50	“_” on page E-50						

A

B

C

CPWG
CPWOC
CPWSC
CPWSUB
CPWTL
CPWTLFG
CTL
C_Model
Chain
Chebyshev
Connector
CostIndex
Crossover

D

DC
DF
DFDevice1
DFDevice2
DF_DefaultInt
DF_Value
DF_ZERO_OHMS
DICAP
DILABMLC
DOE
DRC
DefaultValue
DeviceIndex
Diode

E

EE_BJT2
EE_FET3
EE_HEMT1
EE_MOS1
ETAPER
Elliptic

F

FDD
FINLINE
FSUB

G

GCPWTL
GMSK_Lowpass
GaAs
Gaussian
Goal

H

HB
HP_Diode
HP_FET
HP_FET2
HP_MOSFET
Hybrid

I

IFINL
IFINLT
INDQ
I_Source
InitCond
InoiseBD

J

JFET

K

L

L
LineCalcTest

M

MACLIN
MACLIN3
MBEND

MBEND2
MBEND3
MBSTUB
MCFIL
MCLIN
MCORN
MCROS
MCROSO
MCURVE
MCUREVE2
MGAP
MICAP1
MICAP2
MICAP3
MICAP4
MLANG
MLANG6
MLANG8
MLEF
MLIN
MLOC
MLSC
MLYRSUB
MOS9
MOSFET
MRIND
MRINDELA
MRINDELM
MRINDNBR
MRINDWNR
MRSTUB
MS2CTL
MS3CTL
MS4CTL
MS5CTL
MSABND
MSACTL
MSAGAP
MSBEND

MSCRNR
MSCROSS
MSCTL
MSGAP
MSIDC
MSIDCF
MSLANGE
MSLIT
MSOBND
MSOC
MSOP
MSRBND
MSRTL
MSSLIT
MSSPLC
MSSPLR
MSSPLS
MSSTEP
MSSVIA
MSTAPER
MSTEE
MSTEP
MSTL
MSUB
MSVIA
MSWRAP
MTAPER
MTEE
MTEEO
MTFC
MextramBJT
Mixer
MixerIMT
Multipath
Mutual

N

NodeSet
NoiseCorr2Port

Noisey2Port
Nsample

O

OldMonteCarlo
OldOpt
OldOptim
OldYield
Optim
OptimGoal
Options
OscPort
OutSelector

P

PCBEND
PCCORN
PCCROS
PCCURVE
PCILC
PCLIN1
PCLIN10
PCLIN2
PCLIN3
PCLIN4
PCLIN5
PCLIN6
PCLIN7
PCLIN8
PCLIN9
PCSTEP
PCSUB
PCTAPER
PCTEE
PCTRACE
PC_Bend
PC_Clear
PC_Corner
PC_CrossJunction
PC_Crossover

PC_Gap
PC_Line
PC_OpenStub
PC_Pad
PC_Slanted
PC_Taper
PC_Tee
PC_Via
PIN
PIN2
PLCQ
ParamSweep
PinDiode
PoleZero
Polynomial
Port
PowerBounce
PowerGroundPlane

Q

R

R
RCLIN
RIBBON
RIBBON_MDS
RIND
RWG
RWGINDF
RWGT
RWGTL
R_Model
RaisedCos

S

SAGELIN
SAGEPAC
SBCLIN
SBEND
SBEND2

SCLIN
SCROS
SCURVE
SDD
SL3CTL
SL4CTL
SL5CTL
SLABND
SLCQ
SLCRNR
SLCTL
SLEF
SLGAP
SLIN
SLINO
SLOBND
SLOC
SLOC_MDS
SLOTTL
SLRBND
SLSC
SLSTEP
SLTEE
SLTL
SLUCTL
SLUTL
SMITER
SOCLIN
SPIND
SS3CTL
SS4CTL
SS5CTL
SSACTL
SSCLIN
SSCTL
SSLANGE
SSLIN
SSSPLC
SSSPLR

SSSPLS
SSSUB
SSTEP
SSTFR
SSTL
SSUB
SSUBO
S_Param
S_Port
ScheduleCycle
Short
Substrate
SweepPlan
SwitchV
SwitchV_Model

T

TAPIND1
TFC
TFC_MDS
TFR
TFR_MDS
TL
TLIN
TLIN4
TLINP
TLINP4
TL_New
TQAVIA
TQCAP
TQFET
TQFET2
TQIND
TQRES
TQSVIA
TQSWH
TQTL
Tran

U

UFINL
UFINLT
Unalter

V

VBIC
VIA
VIA2
V_Source
VnoiseBD

W

WIRE
WIRE_MDS

X

Y

Y_Port
Yield
YieldOptim
YieldSpec
YieldSpecOld

Z

Z_Port

__

__fdd
__fdd_v

_

_ac_state
_c1
_c10
_c11
_c12
_c13
_c14
_c15

_c16
_c17
_c18
_c19
_c2
_c20
_c21
_c22
_c23
_c24
_c25
_c26
_c27
_c28
_c29
_c30
_c4
_c5
_c6
_c7
_c8
_c9
_dc_state
_default
_discrete_density
_divn
_freq1
_freq10
_freq11
_freq12
_freq2
_freq3
_freq4
_freq5
_freq6
_freq7
_freq8
_freq9
_gaussian

_gaussian_tol
_get_fnom_freq
_get_fund_freq_for_fdd
_harm
_hb_state
_i1
_i10
_i11
_i12
_i13
_i14
_i15
_i16
_i17
_i18
_i19
_i2
_i20
_i21
_i22
_i23
_i24
_i25
_i26
_i27
_i28
_i29
_i3
_i30
_i4
_i5
_i6
_i7
_i8
_i9
_lfsr
_mvgaussian
_mvgaussian_cov
_n_state

_nfmin
_p2dInputPower
_phase_freq
_pwl_density
_pwl_distribution
_randvar
_rn
_shift_reg
_si
_si_bb
_si_d
_si_e
_sigproc_state
_sm_state
_sopt
_sp_state
_sv
_sv_bb
_sv_d
_sv_e
_tn
_to
_tr_state
_tt
_uniform
_uniform_tol
_v1
_v10
_v11
_v12
_v13
_v14
_v15
_v16
_v17
_v18
_v19
_v2
_v20

_v21
_v22
_v23
_v24
_v25
_v26
_v27
_v28
_v29
_v3
_v30
_v4
_v5
_v6
_v7
_v8
_v9
_xcross

a

abs
access_all_data
access_data
aele
and
arcsinh
arctan
atan2
awg_dia

b

bin
bitseq
boltzmann
by

c

c0
complex
conj

cos
cos_pulse
cosh
cot
coth
coupling
ctof
ctok
cxform

d

d_atan2
damped_sin
db
dbm
dbmtoa
dbmtov
dbmtow
dbpolar
dbwtow
dcSourceLevel
deembed
define
deg
delay
dep_data
deriv
discrete
distcompname
doe
doeindex
dphase
dsexpr
dstoarray

e

e
e0
echo
else

elseif
end
endif
equals
erf_pulse
eval_poly
exp
exp_pulse

f

file
fread
freq
freq_mult_coef
freq_mult_poly
ftoc
ftok

g

gauss
gaussian
generate_gmsk_iq_spectra
generate_gmsk_pulse_spectra
generate_piqpsk_spectra
generate_pulse_train_spectra
generate_qam16_spectra
generate_qpsk_pulse_spectra
get_array_size
get_attribute
get_block
get_fund_freq
get_max_points
global
globalnode
ground

h

hugereal

i

j

k

l

log
logNodesetScale
logRshunt
log_amp
log_amp_cas

m

mag
makearray
max
mcTrial
mcindex
min
model
multi_freq

n

names
nested
nf
nfmin
no
nodoe
noisefreq
noopt
norm
nostat
not
notequals

o

omega
opt
optIter
or

p

parameters
phase
phase_noise_pwl

phasedeg
phaserad
planck
polar
polarcpx
ppt
pulse
pwl
pwlr

q

qelectron
qinterp

r

rad
ramp
randtime
rawtoarray
read_data
read_lib
readdata
readlib
readraw
real
rect
rem
ripple
rms
rn
rpsmooth

s

scalearray
sens
setDT
sffm
sgn
sin
sinc

sine
sinh
sink
sopt
sourceLevel
sprintf
sqrt
ssfreq
stat
step
strcat
stypexform
sym_set
system

t

tan
tanh
temp
tempkelvin
thd
then
time
timestep
tinyreal
to
toi
tranorder
transform

u

u0
unconst
unicap
uniform

v

v
value
vlsb

Index