Agilent Technologies

IC-CAP 2012.01
January 2012
Simulation

Acknowledgments

Errata The IC-CAP product may contain references to "HP" or "HPEESOF" such as in file names and directory names. The business entity formerly known as "HP EEsof" is now part of Agilent Technologies and is known as "Agilent EEsof." To avoid broken functionality and to maintain backward compatibility for our customers, we did not change all the names and labels that contain "HP" or "HPEESOF" references.

Warranty The material contained in this documentation is provided "as is", and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied, with regard to this manual and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Agilent and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

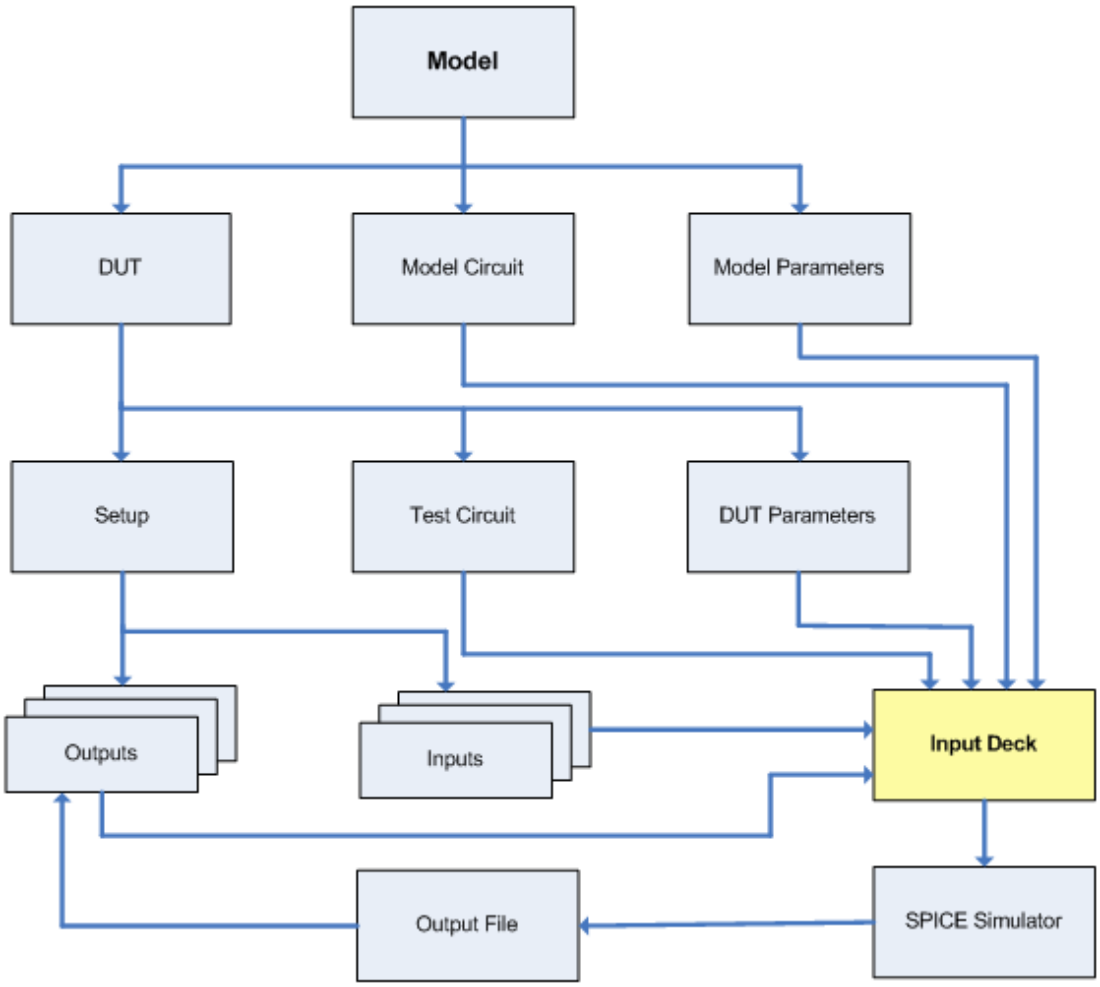
Restricted Rights Legend U.S. Government Restricted Rights. Software and technical data rights granted to the federal government include only those rights customarily provided to end user customers. Agilent provides this customary commercial license in Software and technical data pursuant to FAR 12.211 (Technical Data) and 12.212 (Computer Software) and, for the Department of Defense, DFARS 252.227-7015 (Technical Data - Commercial Items) and DFARS 227.7202-3 (Rights in Commercial Computer Software or Computer Software Documentation).

IC-CAP Simulation Overview

Simulation in IC-CAP is the process of generating device and circuit output characteristics based on an available model. The accuracy of a model determines how simulated characteristics agree with the real physical behavior of devices and circuits.

The following figure illustrates the IC-CAP simulation process.

Figure: Simulation Flow Diagram



You can use five types of simulators in IC-CAP:

- Advanced Design System (ADS) simulator
- SPICE simulators
- HSPICE simulator
- Saber simulator

For details on types of simulators you can use with IC-CAP, refer to *Simulators in IC-CAP* (simulation).

Notes

1. The PC version of IC-CAP supports ADS version 2002 or newer. Older versions of ADS cannot be used with the PC version of IC-CAP.
2. Simulators are provided with IC-CAP as a courtesy and are not supported by Agilent Technologies except the ADS simulator.

Simulation Interfaces

Simulation Interfaces, also known as *templates*, are provided with the simulators. The IC-CAP simulator interface is an open system. It allows you to add any simulator which is similar to one of the provided templates. For details on adding a simulator, refer to *Adding a Simulator* (simulation) section.

The following table lists the IC-CAP Supported Simulators and Corresponding Template Names.

Simulator	Template Name
UCB SPICE2G.6	spice2
UCB SPICE3E2	spice3
HPSPICE	hpspice
HSPICE	hspice
ELDO	eldo
Saber	saber
SPECTRE	spectre (native circuit syntax) spectre443 (spice circuit syntax)
Advanced Design System (ADS)	hpeesofsim (native circuit syntax)

Simulation Types

IC-CAP supports eight basic types of simulation which can be categorized as:

- Standard Simulation
- Special Simulation

Standard Simulation

Standard simulation types are available in the SPICE simulators and include the following:

- DC simulation
- AC simulation
- Transient simulation
- Noise simulation

Special Simulation

The special simulation types include:

- Capacitance Voltage (CV) simulation
- 2-Port (S,H,Y,Z,K,A parameter) simulation
- Multiport (S parameter) simulation
- Time-Domain Reflectometry (TDR) simulation
- Harmonic Balance simulation

Note
Special simulations are not directly available in the SPICE simulators. IC-CAP builds an additional circuitry required in the simulator input files to perform the simulation.

Running a simulation with input and output specifications that do not match with the simulation types could result in the following error:

```
ERROR: Unable to simulate.
Check the Input and Output specifications.
```

The simulators interfaced with IC-CAP may only support a subset of these simulation types and not all of the analysis types available in a particular simulator. For example, non-electrical analysis for Saber is not supported, and Harmonic Balance is supported only with the ADS simulators.

For detailed information on the types of simulation that each simulator supports, refer to the following topics:

- *SPICE Simulators* (simulation)
- *SPECTRE Simulator* (simulation)
- *Saber Simulator* (simulation)
- *ADS Simulator* (simulation)

See Also

Simulation Input and Output Requirements (simulation)

Simulation Input and Output Requirements

This section describes the input and output specifications required for a valid setup for each simulation type and the corresponding measurement types.

DC Simulation

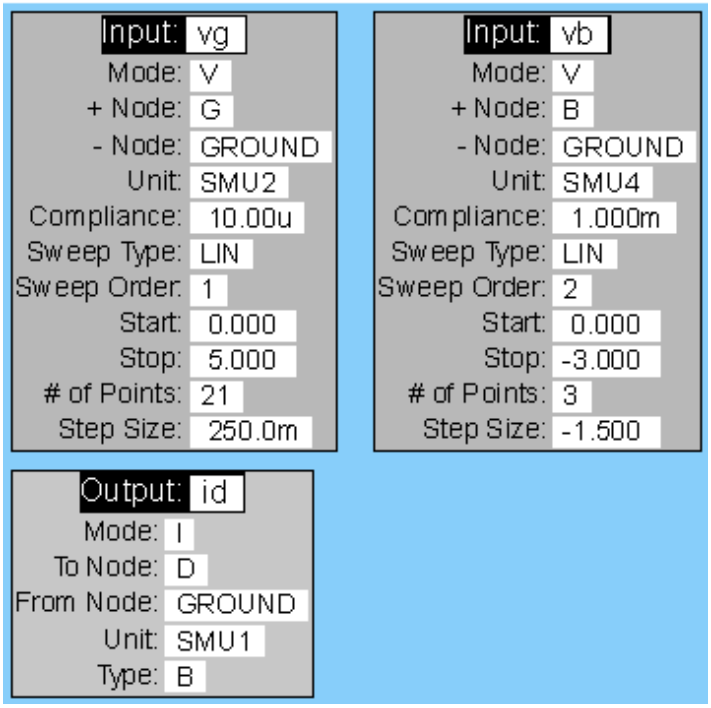
The following table describes the input and output specifications required for a valid setup for DC simulation.

INPUT MODE	VALID SWEEPS	VALID OUTPUTS
V	LIN, LOG, LIST, SYNC, CON	V, I
I	LIN, LOG, LIST, SYNC, CON	V, I

The following figure shows an example of input and output specifications for a MOSFET *id versus vg* setup.

Figure: Example Input and Output, DC Simulation

Active Setup: /nmos2/large/idvg



AC Simulation

The following table describes the input and output specifications required for a valid setup for AC simulation.

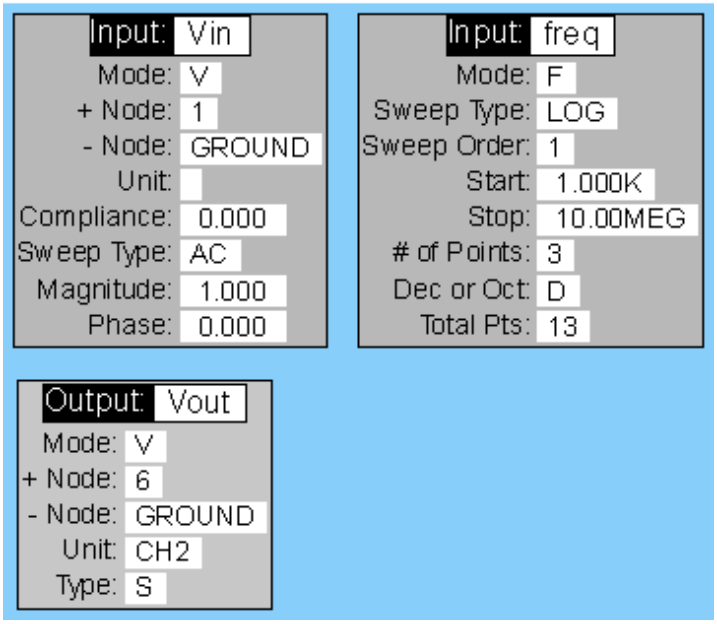
INPUT MODE	VALID SWEEPS	VALID OUTPUTS	FOOTNOTES
V	LIN, LOG, LIST, SYNC, CON, AC	V, I	1, 2, 3
I	LIN, LOG, LIST, SYNC, CON, AC	V, I	1, 2, 3
F	LIN, LOG, LIST, CON	V, I	

1. Exactly one frequency sweep required.
2. At least one AC source required.
3. SYNC is not a valid sweep type when using ADS simulators.

The following figure shows an example of input and output specifications for simulating the output voltage versus frequency of an inverting operational amplifier.

Figure: Example Input and Output, AC Simulation

Active Setup: /opamp1/inv_amp/B_P_macro



Transient Simulation

The following table describes the input and output specifications required for a valid setup for Transient simulation.

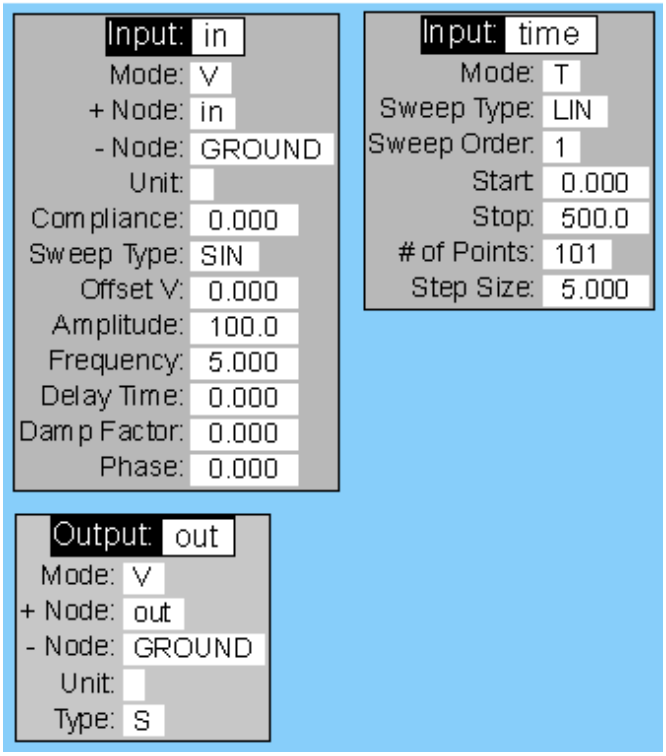
INPUT MODE	VALID SWEEPS	VALID OUTPUTS	FOOTNOTES
T	LIN, LIST, CON	V, I	1, 2
V	LIN, LOG, LIST, SYNC, CON, EXP, PULSE, PWL, SFFM, SIN	V, I	1, 2
I	LIN, LOG, LIST, SYNC, CON, EXP, PULSE, PWL, SFFM, SIN	V, I	

1. Exactly one time sweep required.
2. LIST sweep not supported with Saber.

The following figure shows an example of input and output specifications for voltage versus time characteristics of a differential pair.

Figure: Example Input and Output, Transient Simulation

Active Setup: /nmos2/large/idvg



Noise Simulation

The following tables describe the input and output specifications required for a valid setup for Noise simulation depending on the simulator being used.

Table: Input and Output Requirements for a Noise Simulation using SPICE Simulators

INPUT MODE	VALID SWEEPS	VALID OUTPUTS	FOOTNOTES
V	LIN, LOG, LIST, SYNC, CON, AC	N	1, 2, 3
I	LIN, LOG, LIST, SYNC, CON, AC	N	1, 2, 3
F	LIN, LOG, CON	N	1, 2, 3

1. Exactly one noise output required.
2. Exactly one frequency sweep required.
3. At least one AC source required.

Table: Input and Output Requirements for a Noise Simulation using ADS Simulators

INPUT MODE	VALID SWEEPS	VALID OUTPUTS	FOOTNOTES
V	LIN, LOG, LIST, SYNC, CON, AC	N, V, I	1, 2, 3
I	LIN, LOG, LIST, SYNC, CON, AC	N, V, I	1, 2, 3
F	LIN, LOG, SEG, CON	N, V, I	1, 2, 3

1. There can be multiple outputs.
2. Exactly one frequency sweep required.

3. Both V and I Outputs are DC outputs.

CV Simulation

The following table describes the input and output specifications required for a valid setup for CV simulation.

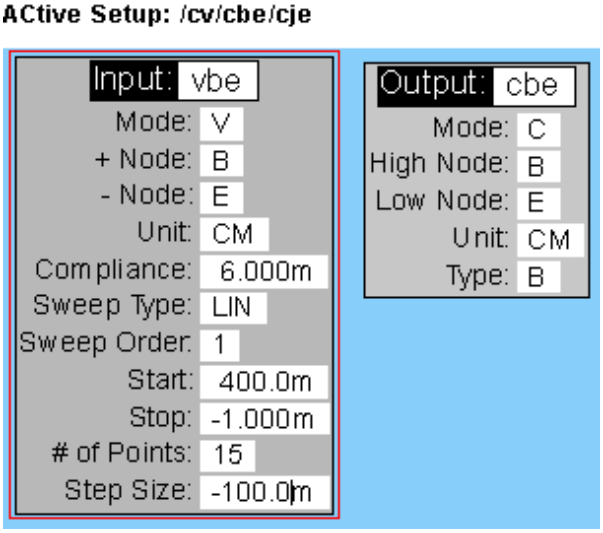
> **Note**
> The frequency at which the CV simulation is performed can be specified using the System Variable CV_FREQ in Hz. If this variable is not specified, the simulation is performed at 1-MG (Hz).

INPUT MODE	VALID SWEEPS	VALID OUTPUTS	FOOTNOTES
V	LIN, LOG, LIST, SYNC, CON	a single C, a single G, a single R, a single X, C + G, C + R	1, 2
I	LIN, LOG, LIST, SYNC, CON	a single C, a single G, a single R, a single X, C + G, C + R	1, 2

1. Any single output type of C, G, R, or X; or exactly one C and one G; or exactly one C and one R.
2. A single C or a C + G is the capacitance using Cp-Gp mode, while a C + R simulates Cs-Rs data

The following figure shows an example of input and output specifications for a BJT base-emitter pn-junction capacitance *versus* voltage setup.

Figure: Example Input and Output, CV Simulation



2-Port Simulation

The following tables describe the input and output specifications required for a valid setup for 2-port simulation.

Table: Input and Output Requirements for a 2-Port Simulation

INPUT MODE	VALID SWEEPS	VALID OUTPUTS	FOOTNOTES
F	LIN, LOG, LIST, CON	S, H, Y, Z, K, A, F	1, 2, 3, 4, 5
V	LIN, LOG, LIST, SYNC, CON	S, H, Y, Z, K, A, F	1, 2, 3, 4, 5
I	LIN, LOG, LIST, SYNC, CON	S, H, Y, Z, K, A, F	1, 2, 3, 4, 5

1. Exactly one frequency sweep required.
2. Exactly one 2-port output (S,H,Y,Z,K, or A) required.
3. Only ADS supports F output. (2-Port noise simulation supports ADS only. If the SIMULATOR is not equal to hpeesofsim, the following error message appears. *Error: in "Output xxxx" High frequency noise output is not supported with current simulator.*)
4. Exactly one 2-port output required for F output.
5. F output can be multiple outputs.

Table: High Frequency Noise Output and Its Data Type Description

F Output	Name (Output Editor)	Symbol shown on Setup Page	Description	Port Input Requirement
Mode	High Frequency Noise	F	High frequency noise mode type	
Data Type	Noise Figure	NF	Noise figure data	Yes,[1, 2]
	Gamma Opt	GAMMAOPT	Optimum source reflection coefficients	No
	Equivalent R Noise	RN	Equivalent noise resistance data	No
	Min Noise Figure	NFMIN	Minimum noise figure data	No
	Equivalent Noise Temperature	TE	Equivalent Noise Temperature data	Yes [1, 2]

1. The port field of the NF/TE noise parameter must not be blank. If the port field *set NF/TE type* is blank, the following error message appears:
Error: in "Output xxxx"
Blank output node name for NF specified.

2. The port name of the NF/TE parameter must be consistent with the port name of the 2-port output.
The port name of the NF/TE output noise must be equal to one of the port names of the 2-port output; otherwise, the following error message appears:
Error: in "Output xxxx"
Port "xx" is not consistent with the 2-port specification node: "xx" or "xx"

The following figure shows an example of input and output specifications for an H21-parameter *versus Vbe* setup.

Figure: Example Input and Output, 2-Port Simulation

Active Setup: /pnp/ac/h21vsvbe

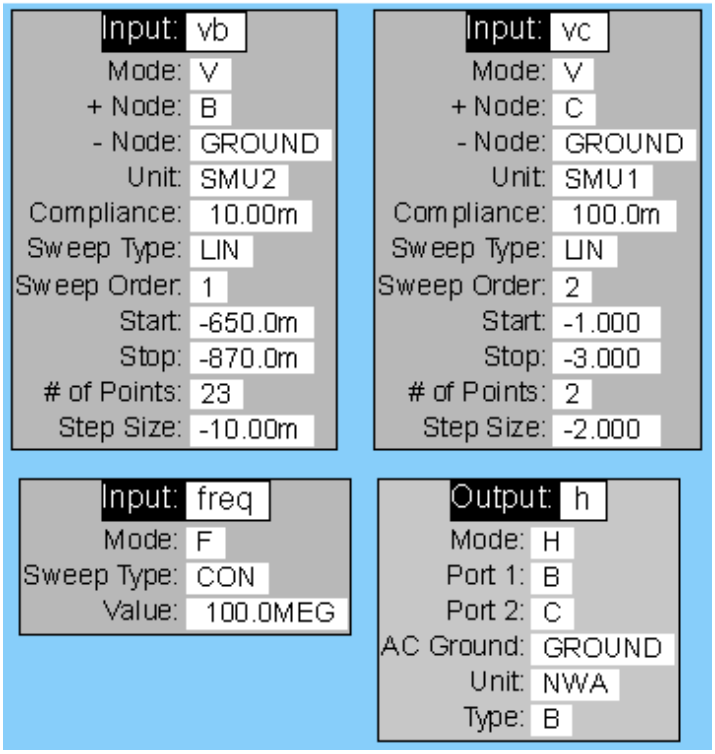
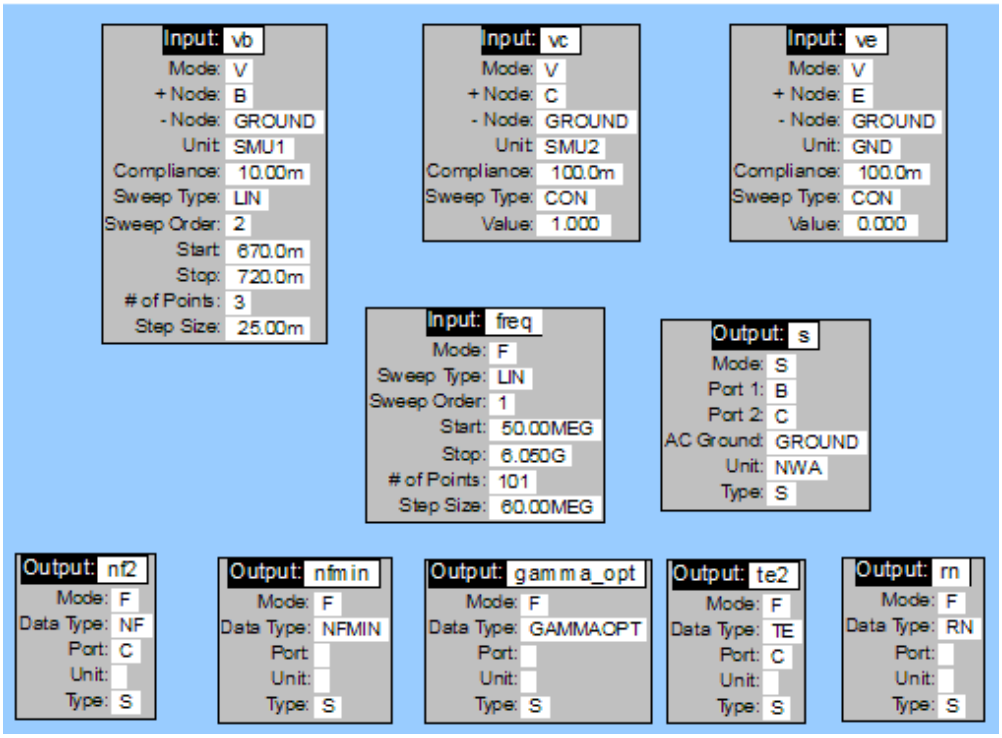


Figure: Example Input and Output, 2-Port Noise Simulation

Active Setup: /bjt_ncehf/sp/sparm_noise_bias



2-Port Circuits

An L-network of LCR is added to port 1 and port 2 to uncouple an AC signal from a DC bias to simulate a DUT using AC analysis. To see an actual input circuit deck, use the

Simulator Debugger.

When the port 1 has an AC source, its signal goes through an R whose value is defined by TWOPORT_Z0 and its default is 50 [ohm]. Then this signal is given to a port 1 through a C whose value is defined by TWOPORT_C, and its default is 100 [F]. The port 1 is also connected to a DC bias source through an L whose value is defined by TWOPORT_L and its default is 100 [H]. The port 2 has a similar L-network whose AC source is replaced by a short to ground.

Note
As the default C and L values are so large compared to actual DUT values, sometimes it is necessary to specify smaller values to reduce numeric errors in simulation. For example, 1mF for C and 1mH for L are more realistic values.

There are two circuits in a single deck to represent two cases where the port 1 has a source and the port 2 has a source. These circuits are generated and added to a DUT for simulators.

Multiport Simulation

Multiport simulations are currently only supported with the ADS simulator (hpeesofsim). This simulation type allows for 3 and 4 port S parameter simulations.

The following table describes the input and output specifications required for a valid setup for multiport simulation in ADS.

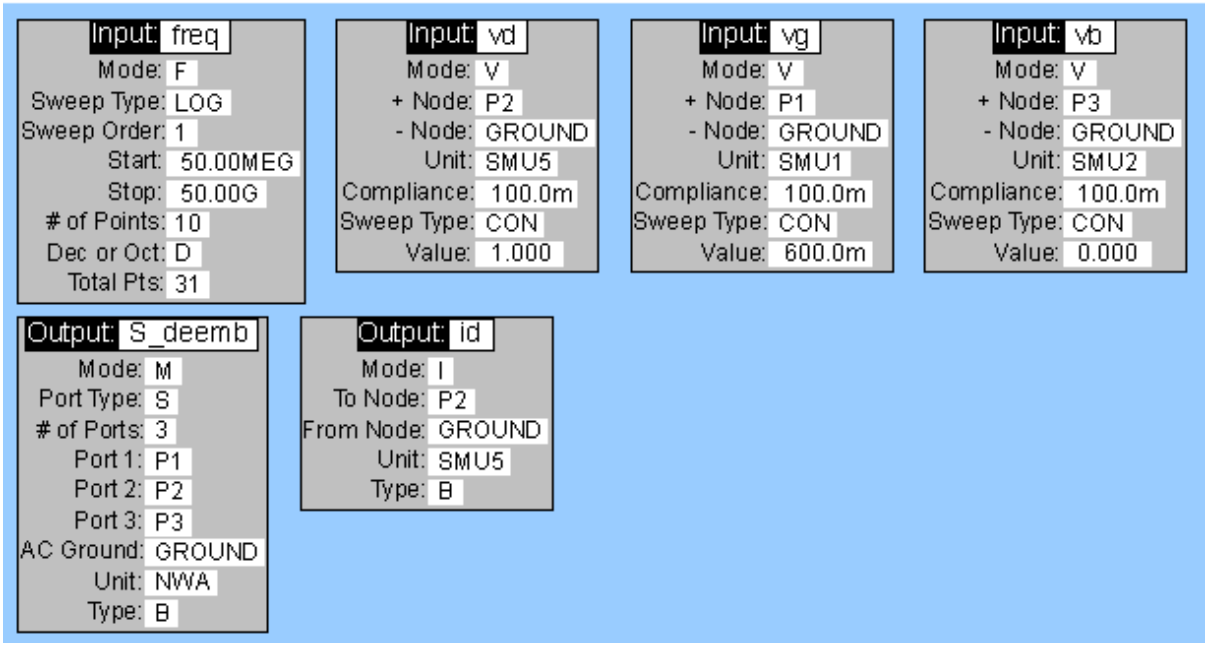
INPUT MODE	VALID SWEEPS	VALID OUTPUTS	FOOTNOTES
F	LIN, LOG, LIST, CON	M	1, 2, 3
V	LIN, LOG, LIST, SYNC, CON	M	1, 2, 3
I	LIN, LOG, LIST, SYNC, CON	M	1, 2, 3

1. Exactly one frequency sweep required.
2. Exactly one multiport output M required.
3. **# of Ports** can be 3 or 4.

The following figure shows an example of input and output specifications.

Figure: Example Input and Output, Multiport Simulation

MultiPortSparSimul_Test.mdl Active Setup: /port3/Spar_MultiPort/Spar_MultiPort



TDR Simulation

The following table describes the input and output specifications required for a valid setup for TDR (Time Domain Reflect) simulation.

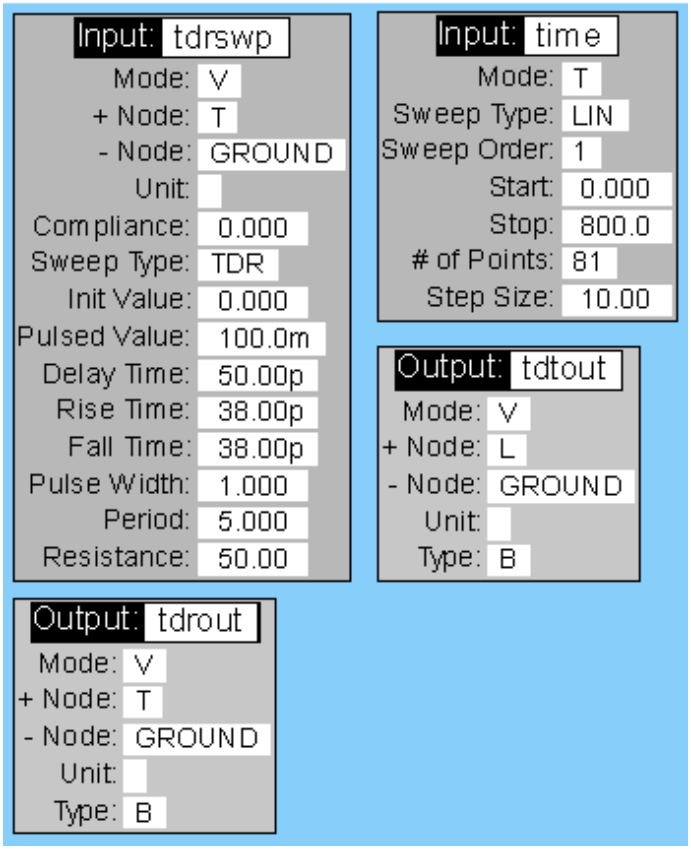
INPUT MODE	VALID SWEEPS	VALID OUTPUTS	COMMENTS
T	LIN, LIST, CON	V	1, 2, 3
V	LIN, LOG,L IST, SYNC, CON, TDR	V	1, 2, 3
I	LIN, LOG, LIST, SYNC, CON	V	1, 2, 3

1. Exactly one time sweep required.
2. Only voltage outputs allowed.
3. Exactly one voltage sweep of type TDR required.

The following figure shows an example of input and output specifications for simulating the reflected and transmitted signal of a simple TDR circuit.

Figure: Example of Input and Output Specifications for a TDR Simulation

Active Setup: /noise/_demo/noise_nalysis/rb_swp



Harmonic Balance Simulation

The following table describes the input and output specifications required for a valid setup for Harmonic Balance simulation.

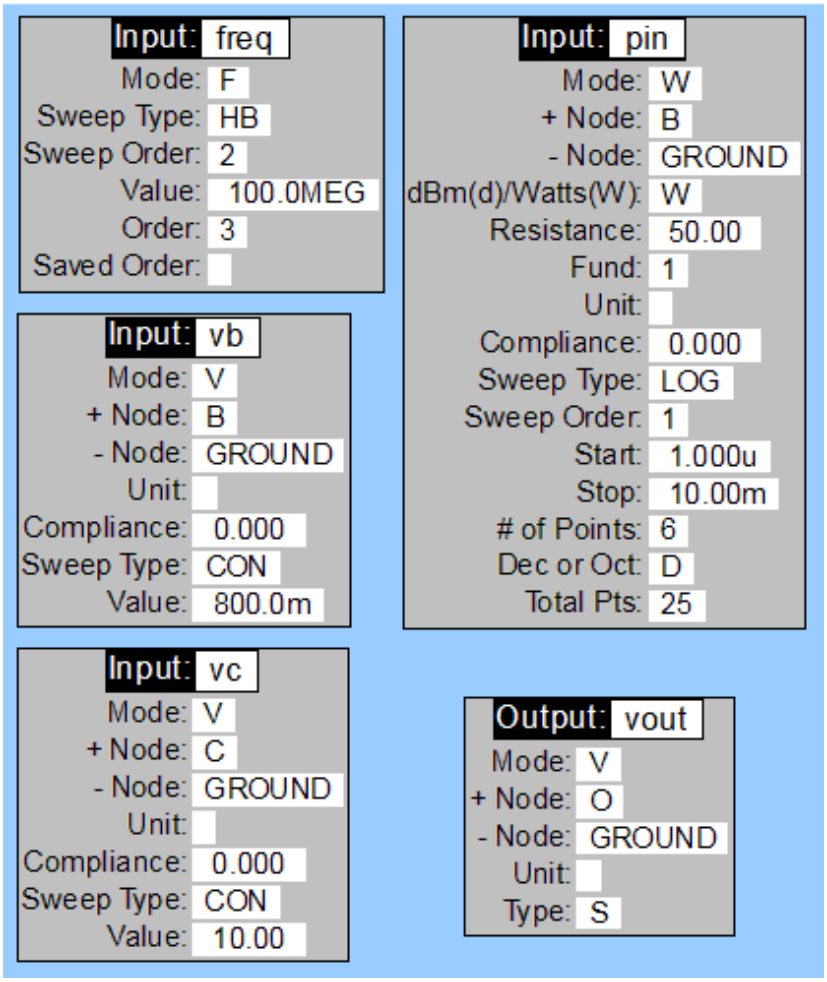
INPUT MODE	VALID SWEEPS	VALID OUTPUTS	COMMENTS
V	LIN, LOG, LIST, SYNC, CON, AC	V, I	1, 2, 3, 4
I	LIN, LOG, LIST, SYNC, CON, AC	V, I	1, 2, 3, 4
F	HB	V, I	1, 2, 3, 4, 5
W (Power)	LIN, LOG, LIST, CON	V, I	1, 2, 3, 4

1. Exactly one frequency sweep with Sweep Type = HB required.
2. Exactly one AC source required: V or I (AC), or W.
3. The Test Circuit includes elements such as DCFEED or DCBLOCK.
4. Units for the Power (W) source can be set to dBm(d) or Watts (W).
5. HB Sweep Type provides 3 fields, **Value:** the fundamental, **Order:** number of harmonics, and **Saved Order:** if blank, DC and all harmonics (that is, Order+1) is read back from simulator. Setting this to a lower value (N) reads only DC and the first N harmonics.

The following figure shows an example of input and output specifications for a BJT power-in versus power-out setup.

Figure: Example of Input and Output Specifications for a Harmonic Balance Simulation

hpsimvbic.mdl Active Setup: /vbic_npn/hb/Harmonic_Balance
hpsimvbic.mdl Active Setup: /vbic_npn/hb/Harmonic_Balance



2 tone Harmonic Balance

The 2 tone Harmonic Balance is currently only supported with the ADS simulator `hpeesofsim`. The following table describes the input and output specifications required for a valid setup for 2 tone Harmonic Balance simulation.

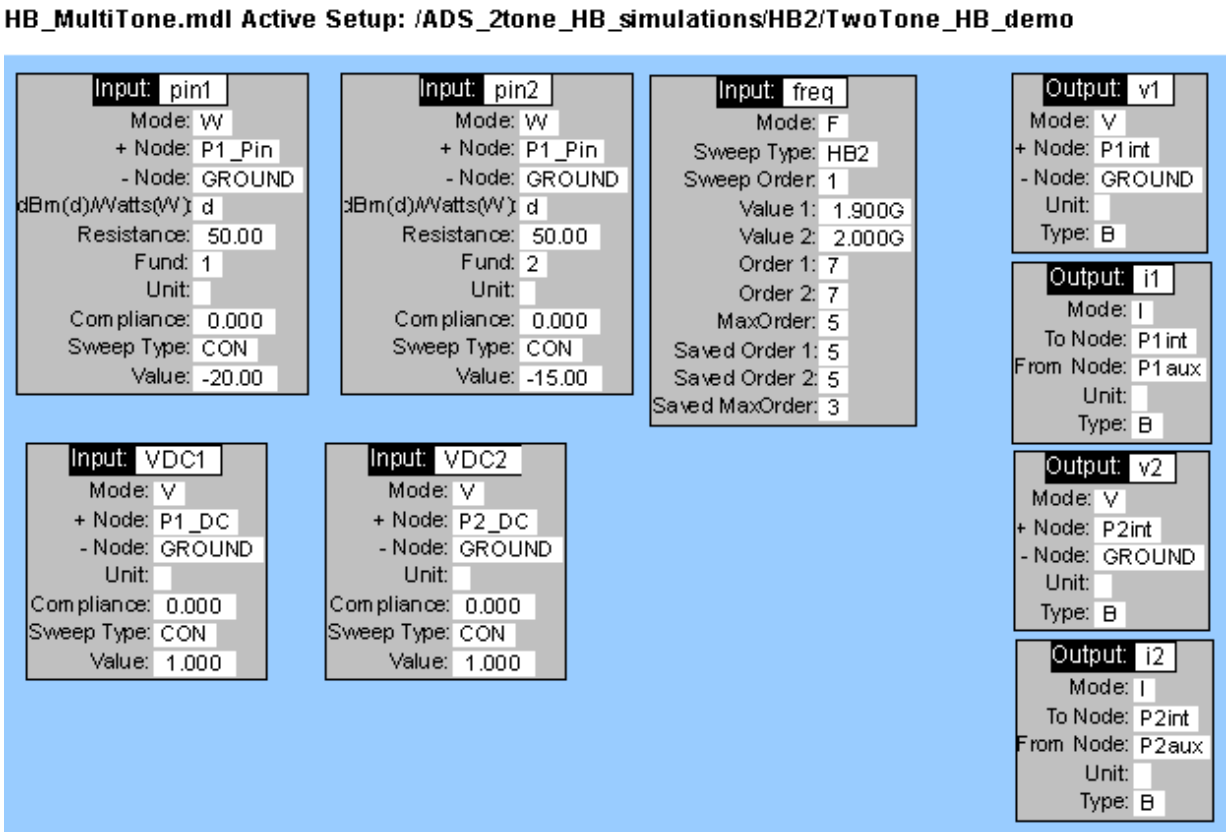
INPUT MODE	VALID SWEEPS	VALID OUTPUTS	COMMENTS
V	LIN, LOG, LIST, SYNC, CON	V, I	
I	LIN, LOG, LIST, SYNC, CON	V, I	
F	HB2	V, I	1, 6
W (Power)	LIN, LOG, LIST, CON, SYNC, LSYNC	V, I	2, 3, 4, 5

1. Exactly one frequency sweep with Sweep Type = HB2 required.
2. Two Power sources with type W required, the setting field should be equal in these two sources.
3. Two Power sources can be in nested sweeps, master-sync or master-lsync pairs.
4. The Test Circuit includes elements such as DCFEED or DCBLOCK.
5. Units for the Power (W) source can be set to dBm(d) or Watts (W).
6. HB2 Sweep Type provides 8 fields.

- **Value 1:** the fundamental for the first tone
- **Value 2:** the fundamental for the second tone
- **Order 1:** number of harmonics of the first tone
- **Order 2:** number of harmonics of the second tone
- **Saved Order 1:** if blank, DC and all harmonics (that is, Order1+1) is read back from simulator. Setting this to a lower value (N) reads only DC and the first N harmonics.
- **Saved Order 2:** if blank, DC and all harmonics (that is, Order2+1) is read back from simulator. Setting this to a lower value (N) reads only DC and the first N harmonics.
- **maxOrder:** determines how many mixing products are to be included in a 2 tone simulation
- **Saved maxOrder:** if blank, all mixing products specified in maxOrder is read back from simulator. Setting this to a lower value (N) reads first N mix products.

The following figure shows an example of input and output specifications.

Figure: Example of Input and Output Specifications for a 2 tone Harmonic Balance Simulation



Performing a Simulation

The steps to perform a simulation include:

1. Selecting a Simulator
2. Specifying Inputs and Outputs
3. Connecting Nodes
4. Specifying Parameter, Variable or Text Sweeps
5. Running a Simulation
6. Aborting a Simulation

Selecting a Simulator

This section describes how you can set a simulator to begin with IC-CAP simulation.

The simulator can be set in one of the following three ways:

- Specify a default startup simulator
- Specify a simulator for a specific model, DUT, or setup
- Specify a simulator using a command

Default Startup Simulator

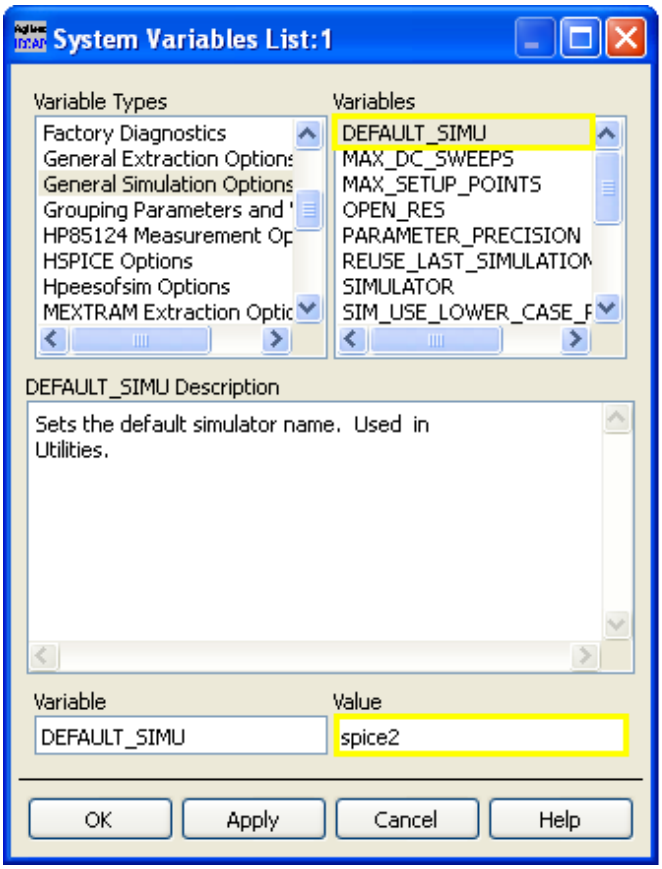
You can specify a simulator as the default simulator on startup by setting the `DEFAULT_SIMU` variable to one of the simulators. This setting is only effective if set at the global level. It is overridden if a different simulator is specified by setting a `SIMULATOR` variable or selecting a simulator through the **Select Simulator** command. When you exit the program, the `DEFAULT_SIMU` setting is saved in the *.icconfig* file.

If this variable is not defined, the default simulator on startup is **spice2**.

To specify a default startup simulator:

1. In the IC-CAP/Main window, select **Tools** > **System Variables**.
2. In the IC-CAP System Variables window, click **System Variables**.
3. In the System Variables dialog box, select **General Simulation Options** as the Variable Type and select the **DEFAULT_SIMU** variable.
4. Enter the simulator name in the **Value** field and click **OK**.

4.



Default Simulation Variable

Enter Simulator Type

Note
You can type the variable name and value in the System Variables window directly without going through the dialog box.

Simulator for a Specific Model, DUT, or Setup

Some models require or perform better with a specific simulator. In these model files, you can specify a simulator for a model, DUT, or setup by setting the `SIMULATOR` variable. This allows you to use different simulators for different models, DUTs or setups, since a `SIMULATOR` variable can be specified at any level.

The following table lists the model files for which the `SIMULATOR` variable is defined.

Table: Model Files with Predefined Simulators

Model File Name	SIMULATOR Value
bjt_ft.mdl	hpspice
bjt_ncehf.mdl	hpspice
hpsimbjt_ncehf.mdl	hpeesofsim
hpsimbjt_nhf.mdl	hpeesofsim
hpsimnpn.mdf	hpeesofsim
hpsimvbic.mdl	hpeesofsim
mxt504_npn.mdl	hpeesofsim
sabernpn.mdl	saber
spectre_ncehf.mdl	spectre

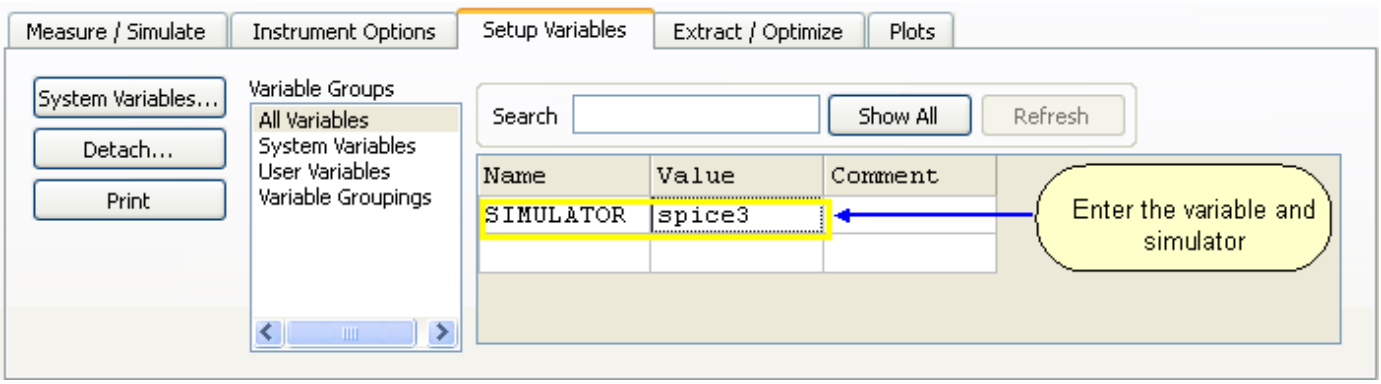
spectrenpn	spectre
pn_diode.mdl	spice2
CGaas1.mdl	hpspice
CGaas2.mdl	hpspice
CGaashf.mdl	hpspice
CGaashfax.mdl	hpspice
hpsimHPEEfet3.mdl	hpeesofsim
UCBGaas.mdl	spice3
UGaashf	spice3
lc.mdl	spice3
sabercirc.mdl	saber
sys110_verify.mdl	hpspice
hnmos6.mdl	hspice
hnmos28.mdl	hspice
hpmos28.mdl	hspice
sabernmos.mdl	saber
noise_simu.mdl	spice3
bjt_1f_noise.mdl	hpspice
mos_1f_noise.mdl	spice3
opamp.mdl	hpspice
bjt_ncehfp.mdl	hpspice
BSIM3_DC_CV_Measure.mdl	spice3
BSIM3_DC_CV_Extract.mdl	spice3
BSIM3_RF_Measure.mdl	spice3
BSIM3_RF_Extract.mdl	spice3
BSIM3_AC_Noise_Tutorial.mdl	spice3
BSIM3_CV_Tutorial.mdl	spice3
BSIM3_DC_Tutorial.mdl	spice3
BSIM3_Temp_Tutorial.mdl	spice3
BSIM3_DC_CV_Finetune.mdl	spice3
BSIM4_DC_CV_Measure.mdl	spice3
BSIM4_DC_CV_Extract.mdl	spice3
BSIM4_RF_Measure.mdl	spice3
BSIM4_RF_Extract.mdl	spice3
BSIM4_DC_CV_Tutorial.mdl	spice3
BSIM4_DC_CV_Finetune.mdl	spice3

When a simulation is performed, IC-CAP looks for the `SIMULATOR` variable first, and if found, IC-CAP makes simulator specified in the variable as the active simulator.

The Select Simulator dialog box changes to reflect the name of the active simulator. If the `SIMULATOR` variable is not defined, IC-CAP uses the simulator displayed in the Select Simulator dialog box. For information on selecting a simulator using Select Simulator dialog box, refer to Specify Simulator Using a Command.

To set a simulator for a specific model, DUT, or setup:

1. Select the appropriate model, DUT, or setup folder and click the corresponding **Variables** tab.
2. Type `SIMULATOR` in an empty variable **Name** field and type the name of the simulator in the corresponding **Value** field.



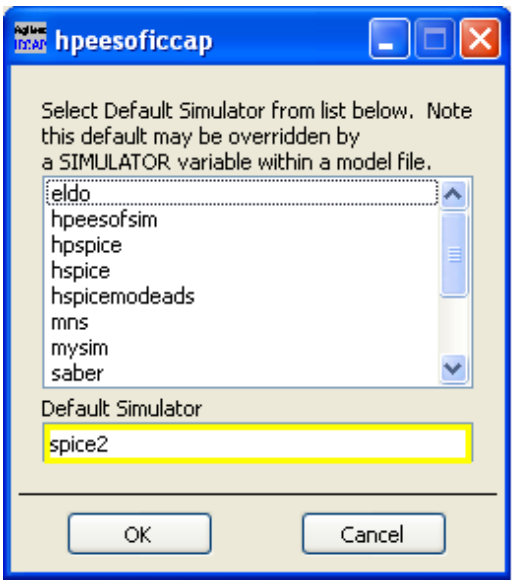
Note
To use a different simulator after one has been specified by the `SIMULATOR` variable, reset the simulator using the **Select Simulator** command.

Simulator Using a Command

The **Select Simulator** command sets the simulator to be used for all simulations performed in the current session, except when simulating a model, DUT, or setup for which a `SIMULATOR` variable has been defined.

To set a simulator without using a variable:

1. In the IC-CAP Main window, choose **Tools > Select Simulator**. A hpeesoficcap dialog box is displayed which list the supported simulators.



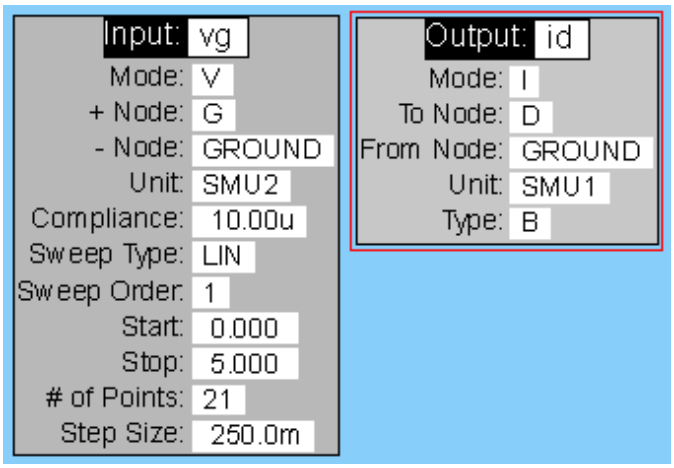
Default active simulator

2. Select a simulator name or type the name of a simulator you have linked with IC-CAP in the **Default Simulator** field and click **OK**.

Specifying Inputs and Outputs

When running a simulation, IC-CAP builds the simulation input deck using the circuit description and the input and output specifications. The circuit description provides the model information. The input and output specifications provide the input stimuli and requested output data, as well as the information needed to determine the type of simulation being performed.

Specifying inputs and outputs is independent of the type of simulation being performed. When specifying input for a simulation, you must enter the sweep mode in the **Mode** field of the Input table, and the node connections. The **Input** field changes, depending on the type of mode specified.



Connecting Nodes

The convention used for node connections when sourcing voltage and current are provided below:

- When you specify an Input Mode of *V*, the **+Node** and *-Node* fields are available in the Input table. In this case, the **+Node** is considered to be the positive side of the voltage source and the **-Node** is the negative side.
- When you specify an Input Mode of *I*, the **To Node** and **From Node** fields are available in the Input table. Current flows from the **From Node** to the **To Node**.

When IC-CAP builds the simulation input deck, it creates the source name by concatenating the mode character, the first three characters of the **+Node (or To Node)** and the first three characters of the **-Node (or From Node)**. These source names are used in the simulation input deck to specify the sweeps and constants. Specified outputs may also reference these names.

Source names are limited to 8-characters. This limit may cause issues in a simulation, if, for example, two inputs are specified as follows:

Mode = V +Node = BASE -Node = EMITTER	Mode = V +Node = BASE -Node = EMITTER2

From this input, IC-CAP creates the same source names: `VBASEMI` and `VBASEMI`. You can

avoid this potential conflict with source names by choosing node names in circuit descriptions carefully. When choosing node names with more than three characters, make sure that the first three characters are unique with respect to the first three characters of any other node names.

When you enter an invalid node name, such as K in any of the input and output node fields and try to simulate, the program sends an error message:

```
ERROR: Invalid Input node name K used
ERROR: Unable to simulate.
Check the Input and Output specifications.
```

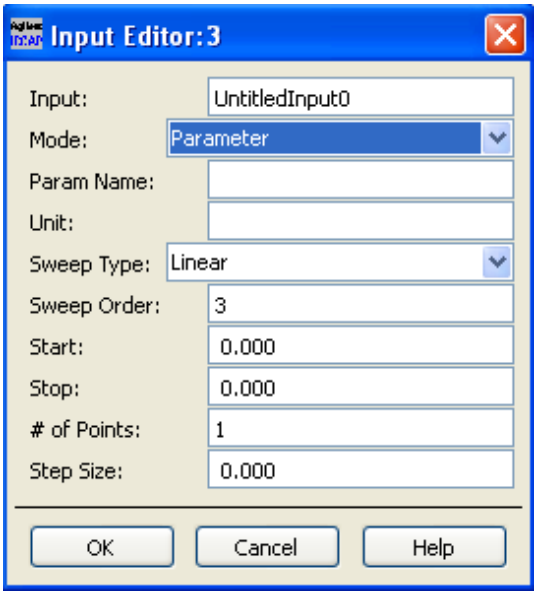
Specifying Parameter, Variable or Text Sweeps

In addition to the list of valid sweep modes for each simulation type listed, you can sweep parameter values and variables. For example, you can generate a family of beta versus IC curves by using the BF parameter of the bipolar transistor model as the step input or you can sweep the operating temperature variable `TEMP` to analyze temperature effects.

Note
You can set the value for a constant or values for sweeps of the simulation temperature by adding the `TEMP` variable to the variable table and creating an input (Mode = Parameter and Name = TEMP) in the setup.

To sweep parameters or variables:

1. In the Model window, select the DUT and the setup.
2. Click **Measure/Simulate**.
3. Click **New Input**. The Input Editor is displayed.



4. In the **Mode** field, select **Parameter** from the drop-down list.
5. Specify a parameter or variable by entering its name in the **Param Name** field.
6. Enter all other necessary information and click **OK**.

A parameter sweep is a valid input mode for all simulation types. Specifying parameter

sweeps may differ for devices and circuits depending on the type of simulator being used.

For an example of simulator-specific parameter sweeps, refer to the following topics:

- *SPICE Parameter Sweeps* (simulation)
- *Saber Parameter Sweeps* (simulation)
- *ADS Parameter Sweeps* (simulation)

Hierarchical Parameter Sweeps

You can perform parameter sweeps when using hierarchical models. However, when you sweep a parameter from a model lower than the level from which the simulation is being performed, you must specify the complete path name of the parameter in the Name field of the input.

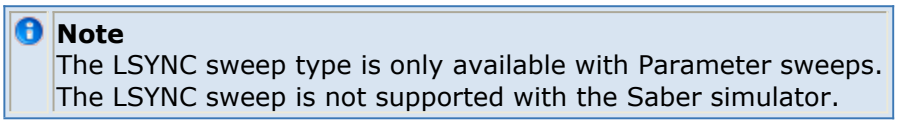
Sychronized List Sweeps (LSYNC)

To synchronize a parameter sweep with other parameter sweeps, set the sweep type to LSYNC (*Synchronized List* when using the *Edit* feature on an input). Just like a normal SYNC sweep, you must specify the name of the master sweep by entering it in the *Master Sweep* field.

Unlike a normal SYNC sweep, the entries are not limited to an offset and a multiplier. The LSYNC sweep enables you to specify an arbitrary list of points. IC-CAP automatically provides the required number of points after the master sweep is set. If the number of points in the master sweep changes, simply click on the LSYNC *Master Sweep* field to update the number of points in the input.

One application of LSYNC sweeps is to simulate an arbitrary collection of device Lengths and Widths.

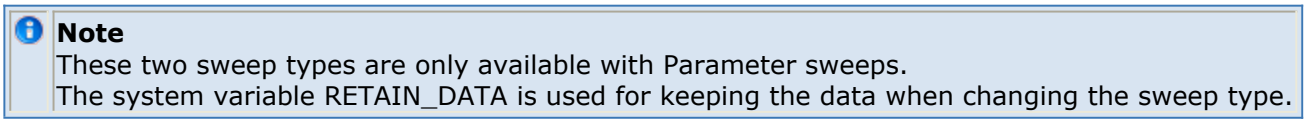
If a master sweep and multiple LSYNC sweeps are saved to an MDM file, they can only be imported into a setup where the same sweeps are either all CON sweeps or the sweeps are synchronized using LSYNC. One sweep cannot be a LIN sweep and another one be a CON sweep, even if that combination exist in the MDM file. To use a LIN sweep with a CON sweep, use LSYNC to synchronize the CON sweep to the LIN sweep and enter the same value for all list points.



Note
The LSYNC sweep type is only available with Parameter sweeps.
The LSYNC sweep is not supported with the Saber simulator.

Specifying Text Sweeps

The sweep type TLIST is a LIST accept string values.
The sweep type TLSYNC is LSYNC accept string values.



Note
These two sweep types are only available with Parameter sweeps.
The system variable RETAIN_DATA is used for keeping the data when changing the sweep type.

Simulating Open Circuits

IC-CAP uses the OPEN_RES variable to handle any floating nodes. This variable allows an open circuit to be simulated as a large resistance. The value of the resistance is equal to the value of the OPEN_RES variable. A resistor of this magnitude is automatically connected to all external circuit nodes not connected to a specified source. When the OPEN_RES variable is not specified, a current source set to zero (0) amps is used instead. However, using the current source may cause simulation convergence problems.

Running a Simulation

You can perform a simulation on an active setup or on all the setups in an active DUT.

Simulating an Active Setup

To perform a simulation on the active setup, select the setup and click **Simulate Setup** icon () in the Model window.

You can also choose **Simulate > Active Setup** from the menu bar of the Model window to perform simulation on an active setup.

Simulating All the Active Setups in a DUT

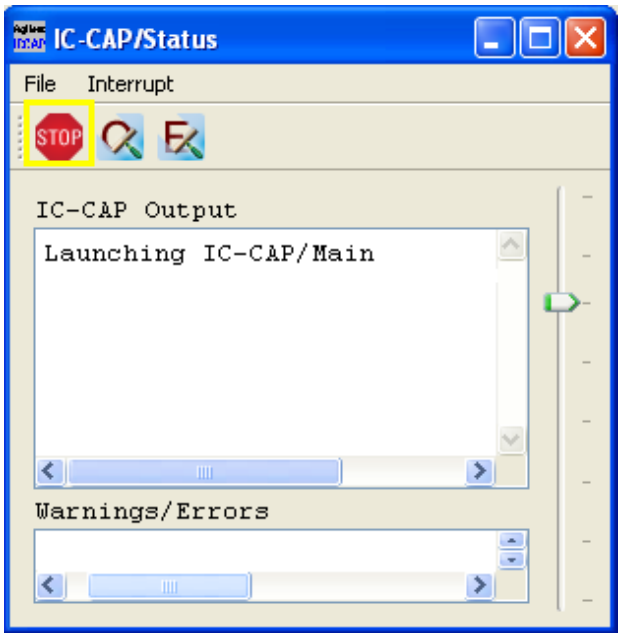
To perform a simulation on all the setups in an active DUT, select the DUT and choose **Active DUT** under **Simulate** menu in the Model window.

Aborting a Simulation

You can abort a running simulation at any time from the IC-CAP Status window.

To abort a simulation:

Click Interrupt IC-CAP Activity icon or choose **Interrupt > IC-CAP Activity** in the IC-CAP/Status window.



After you abort a running simulation, the simulation stops and the following message appears in the Status window:

```
HALTED: Simulation interrupted by user
```

Using Simulation Debugger

The Simulation Debugger is a useful tool for determining why a simulation failed.

When a simulation fails, the program displays an error message:

```
Simulation Failed: Data Unchanged Use Simulation Debugger in Utilities Menu for more information
```

The SPICE-type simulators accept an input deck that contains both the circuit description and analysis commands.

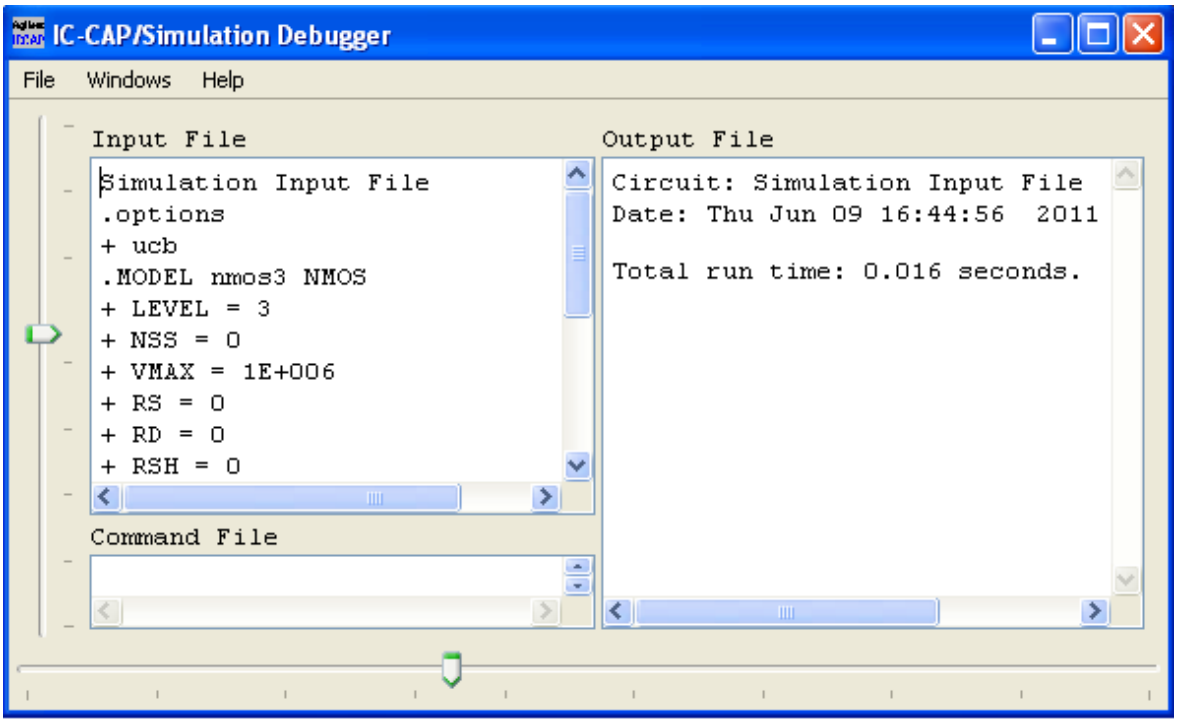
The Saber simulator requires two separate decks. The Saber input deck, displayed in the Input editor, contains the circuit description, written in the MAST modeling language. The Saber command deck, displayed in the Command editor, contains the analysis commands to be performed by the simulator. The Command editor is only used with the Saber simulator.

The input editor displays the input information used in the simulation. For Saber simulations, the command editor information is used also. You can quickly see how changes would affect your results by changing the input (and for Saber, command) files, performing a manual simulation, and observing the results in the output editor. For more information, refer to Using the Manual Simulate Function.

In many cases, the output text file includes the error messages displayed when the simulation fails. You cannot edit an output text file.

To use the Simulation Debugger:

1. In the IC-CAP/Main window, click **Simulation Debugger** icon (). The IC-CAP Simulation Debugger window is displayed. By default, the Input, Command, and Output editors are blank.
2. Initiate the simulation in the Model window. The program sends the input deck and output text files to the input and output editors of the Simulation Debugger.

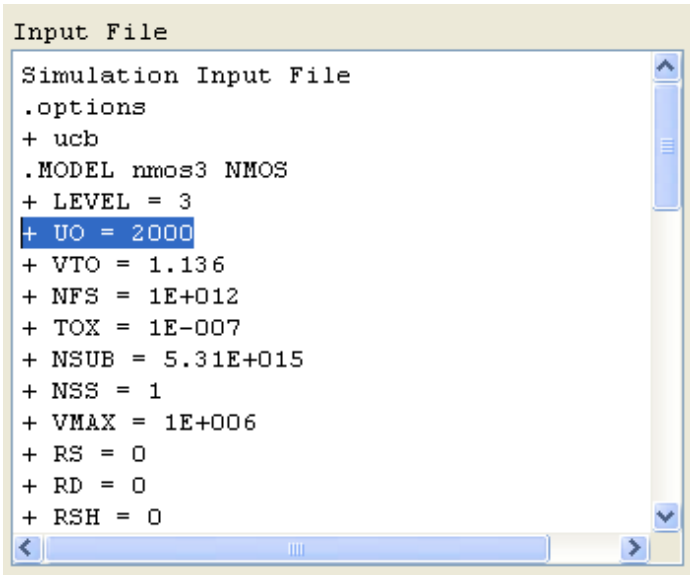
2. 

Manual Simulate Function

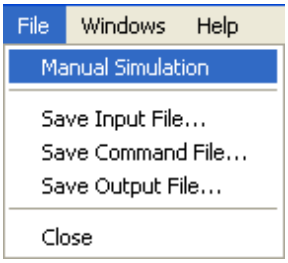
The **Manual Simulate** function simulates the input deck displayed in the **Input Editor**. For example, you can perform a manual simulation after changing some parameter values or sweep values directly in the input file deck, without having to change these values in the IC-CAP Circuit definition, parameter tables, or input and output specifications.

To execute a manual simulation:

1. In the Simulation Debugger window, edit the inputs and command decks displayed in the Input and Command editors panes.



2. Then, select **File > Manual Simulation** in the Simulation Debugger window.

2. 

Saving the Simulation Debugger Files

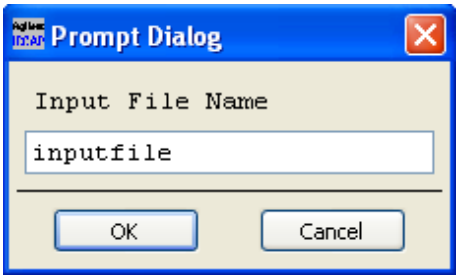
You can save the input, command, and output files displayed in the individual panes of the Simulation Debugger.

To save Simulation Debugger Input, Output, and Command files:

1. Select **File > Save** and choose the appropriate option in the Simulation Debugger window.

   Note
   You can save the Command files for Saber simulation only.

2. In the Input Prompt dialog box, enter a file name.



If you enter a file name only, the file is saved to the current working directory. To save the file to another directory, enter the file name along with the full path.

Linking a Simulator to IC-CAP

The interface for linking a simulator to IC-CAP depends on the type of simulator being used.

A non-piped simulation receives the input deck information from a file, performs the simulation, and sends the binary output data and resulting text output to other files. The simulator process is restarted for every simulation. The non-piped simulations are identical, regardless of simulator type.

The definition of a piped simulation differs for SPICE simulators, Saber simulators, and the ADS simulator. For descriptions of these differences, refer to one of the following:

- *Piped and Non-Piped SPICE Simulations* (simulation)
- *Piped and Non-Piped SPECTRE Simulations* (simulation)
- *Piped and Non-Piped Saber Simulations* (simulation)
- *Piped ADS Simulations* (simulation) and *Non-Piped ADS Simulations* (simulation)

The following simulator links have been tested to work for IC-CAP 2002 PC:

- Remote to spectre on UNIX
- Local to HSPICE on PC
- Local to hpeesofsim on PC
- Local to SPICE2 on PC
- Local to SPICE3 on PC
- Local to HPSPICE on PC

The following simulator links may work for IC-CAP 2002 PC, but were not thoroughly tested:

- Remote to HSPICE on UNIX
- Remote to HSPICE on another PC
- Remote to hpeesofsim in CANNOT_PIPE mode on another PC
- Remote to hpeesofsim in CANNOT_PIPE mode on UNIX

Adding a Simulator

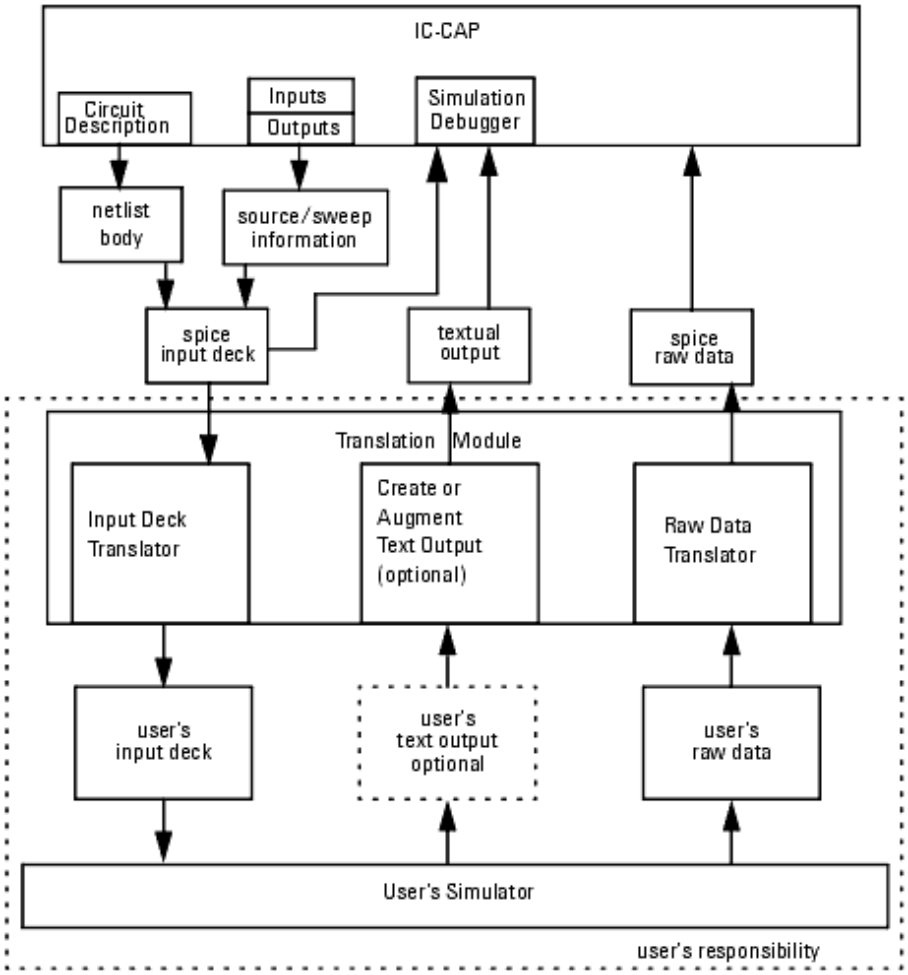
The IC-CAP Open Simulator Interface allows the addition of any simulator to IC-CAP. The Figure: IC-CAP Open Simulator Interface Data Flow Diagram shows a data flow diagram of this interface.

In this design, the IC-CAP system has no knowledge of your simulator. The circuit description is entered in the format corresponding to the template name in the usersimulators field (second argument).

Template Name	Syntax
spice2 spice3 hpspice hspice eldo spectre443 spicemodeads	spice
spectre spmodeads	spectre
saber	saber
hpeesofsim	hpeesofsim (ADS)

These simulators are referred to as the *template* simulators when writing interface code. For example, when you specify a circuit description with syntax matching the syntax of *spice2,* then *spice2* is referred to as the *template* simulator.

IC-CAP Open Simulator Interface Data Flow Diagram



IC-CAP generates the input deck as if the selected simulator were the template simulator. When you enter your simulator in the *Select Simulator* dialog box or the SIMULATOR variable, the input deck is sent through a module of code that you have written. This module of code is referred to as the *Translation Module* in the figure. The Translation Module consists of two translators. The first translates the IC-CAP generated input deck to your simulator's input deck. The deck is then sent to your simulator for analysis. When the simulation is completed, the second translation accepts your simulator's raw data and converts this to the raw data format of the template simulator. This data is returned to IC-CAP for evaluation.

Starting with IC-CAP 2002 PC and IC-CAP 2004 UNIX, you can specify whether the simulator returns the raw data in big-endian or little-endian format. If you do not specify a format, IC-CAP assumes that the data is returned in the order native to the platform, which is big endian for UNIX and little endian for the PC. To specify big endian, append .be or the alias .hpux or .sparc to the template name. To specify little endian, append .le or the alias .pc to the template name. For additional information, see the *README.byteorder* file in the source directory *$ICCAP_ROOT/src* directory.

It is possible to have a circuit description in the native syntax of the simulator that you will use. This is done by using the Open Circuit Parsing Interface that is available in IC-CAP. Though the netlist body is provided in your selected simulator's native syntax, the source and sweep information that IC-CAP adds to the netlist body is still in the template syntax. To use this interface, you must specify the name of the executable responsible for

generating the netlist body.

When the Simulation Debugger is running, the file displayed as the input file is the template simulator's input deck generated by IC-CAP. The Simulation Debugger's output file can be any text file generated in the Translation Module section of code. This allows many options with respect to what information can be included in this text file that may be helpful for debugging purposes.

The Translation Module section of code consists of the two translators, but may also include any other desired functionality. For example, you may read the text file back from your simulator and send this file back to IC-CAP to be displayed in the Simulation Debugger's output file. The Translation Module can also be written to generate debug statements in a text file to show the progress of the input and output translations. This text file can then be sent to IC-CAP and displayed in the Simulation Debugger's output file.

As part of the interface code, you are supplied with a file called *usersimulators* in the directory *$ICCAP_ROOT\iccap\lib*. This file must contain the user-specified information for each simulator added to the IC-CAP system. Five fields of information must be specified when adding a user simulator to IC-CAP. The fields of information must be separated by a space. The fields may or may not be surrounded by quotation marks. A blank, such as *host_name*, is indicated by a pair of quotation marks (""). An optional sixth field is available to use the Open Circuit Parsing Interface.

The general format is as follows:

```
simulator_name template_name path_name host_name pipe_capability
[parser_path] [special_path]
```

where:

simulator_name is the name of the user simulator being added to the list. This is the name you will specify when selecting the simulator in the *Select Simulator* dialog box or the SIMULATOR variable. You may assign any name to this field EXCEPT for any of the reserved IC-CAP template names. The reserved template names are shown in the table *IC-CAP Supported Simulators and Corresponding Template Names* (simulation).

template_name is the name of the *template* simulator. The user-written translation modules map the input file format of the user's simulator to the input file format of the *template* simulator. Likewise, the output file format of the *template* simulator is translated into the output file format of the user's simulator. To specify that the simulator returns the raw data in big endian format, append *.be* or the alias *.hpux* or *.sparc*. To specify that the simulator returns the raw data in little endian format, append *.le* or the alias *.pc*.

path_name is the complete path name of the user's simulator executable file or translation module. Use back slashes when naming the path to a simulator on a PC and forward slashes when naming the path to a simulator on UNIX.

host_name is the host machine name on which the simulator can be used. The

purpose of this information is for remote simulations where only a particular computer is able to access a simulator. If this field is blank, indicated by a pair of quotation marks (""), the simulation is executed on the machine currently running IC-CAP. The format of *host_name* is < *host* > [< *tmp_dir* >], where <*host*> is any host name permitted by *rsh* and *remsh*. Examples include remotebox, remotebox.my.com, 192.168.4.4, and icuser@remotebox. The last form enables users with sufficient permission to simulate to the machine *remotebox* as if the user *icuser* was performing the simulation. This is useful when simulating to a UNIX machine from a PC when the login names for the PC don't match the login names for the UNIX machine. < *temp_dir* > is optional and it enables you to specify a location for IC-CAP's temporary files. The default location is */var/tmp* on the remote machine. For example, if a PC is running services and it meets the requirements in *Network Security* (simulation), */var/tmp* (UNIX notation) may not work for the PC. You can override this by specifying something like _c:\temp_. For more information, see *Remote Simulation* (simulation).

pipe_capability is either CAN_PIPE or CANNOT_PIPE. It specifies whether or not the simulator has the ability to perform piped simulations. When CANNOT_PIPE is specified in this field, a non-piped simulation is done even when the IC-CAP simulation debugger is off.

parser_path is an optional entry that specifies the name of the executable responsible for generating the netlist body and providing IC-CAP with the necessary parameter/node information.

special_path is a simulator-specific field and may have different meanings for each simulator. Currently it is only required by the saber interface and will be ignored for any other simulator template. The field can be completely omitted from all templates but saber. For saber, it should provide the path to the *aimsh* executable in your saber installation. Note, to specify this field without declaring a *parser_path*, you must specify two quotations "" for the *parser_path* field.

Using the Open Circuit Parsing Interface

To use the interface and generate a circuit description in the native syntax of the simulator, you must specify the executable in the *usersimulators* file. This optional field in *usersimulators* is the path to the circuit parser. This enables the simulator link to use your simulator's native syntax in IC-CAP's circuit description shown in the Circuit folder. Without this interface, your circuit must be represented as *spice*, *hpeesofsim*, or *sabre*. If you specify an executable, it is responsible for the following actions:

- Creates a *parsed_file* that IC-CAP will use to merge parameters at each simulation.
- Identifies all pertinent parameters for in the DUT Parameters and Model Parameters folders.
- Identifies the number and names of all nodes that will be used in IC-CAP.

Your parser will be invoked with two arguments, the source file name and the output file name. The parser is responsible for generating the output file which is the same as the input file with substitutions for node names, parameters, and model names. In addition,

the circuit type must be declared.

Determining the circuit type differs depending on the template being used. For any of the *spice* templates, the netlist should consist of one instantiation of one model or a subcircuit.

```
<instance line>
<model card>{code}
```

or

```
.subckt
.
.
.ends
```

The first should be declared a circuittype of the first character of the instance line. The second should be declared a circuit of type *X*. The instance line should be omitted from the output file as IC-CAP will generate this line with the proper node numbers for the type of simulation being performed.

For *hpeesofsim* simulations, the circuit is similar, either a subcircuit, or a model and an instance. circuittype for a subcircuit is still *X*, but for the instance netlist, the type is always *D*.

For *saber* simulations, circuittype is again a *D* for device netlists and an X for subcircuit netlists, but for this template, you must set device type as well which is the actual name of the device type.

The output file should place the token $ where the name of the model should appear in the netlist. It should place the token <name>$ where the value for parameter named <name> should appear.

```
Example Device circuit:
D1  1 = A  2 = C  DIODE
.MODEL DIODE D
+ IS = 1E-14
+ N = 1.0
+ BV = 1000
+ IBV = 1m
+ RS = 0
+ CJO = 0
+ VJ = 1.0
+ M = 0.5
+ FC = 0.5
+ TT = 0
+ EG = 1.110
+ XTI = 3.0
Output File:
.MODEL $modname$ D
+ IS = $pvalIS$
+ N = $pvalN$
+ BV = $pvalBV$
```

```
+ IBV = $pvalIBV$
+ RS = $pvalRS$
+ CJO = $pvalCJO$
+ VJ = $pvalVJ$
+ M = $pvalM$
+ FC = $pvalFC$
+ TT = $pvalTT$
+ EG = $pvalEG$
+ XTI = $pvalXTI$
Example subcircuit circuit
.OPTION gmin=1e-30
.SUBCKT LED 1=A 2=C
RS  1 11 1m
DLO 11 2 DLO
DHI 11 2 DHI
.MODEL DLO D
+ IS  = 1E-29
+ N   = 1
.MODEL DHI D
+ IS  = 1E-34
+ N   = 1
+ CJO = 100p
+ M   = .4
+ VJ  = 2
+ FC  = .5
.ENDS
Output File:
.SUBCKT $modname$ 1 2
RS 1 11 $pvalRS$
DLO 11 2
+ DLO
DHI 11 2
+ DHI
.MODEL DLO D
+ IS = $pvalDLO.IS$
+ N = $pvalDLO.N$
.MODEL DHI D
+ IS = $pvalDLO.IS$
+ N = $pvalDLO.N$
+ CJO = $pvalDLO.CJO$
+ M = $pvalDLO.M$
+ VJ = $pvalDLO.VJ$
+ FC = $pvalDLO.FC$
.ENDS
```

The parser must print the commands to standard output that tell IC-CAP about the circuit it has parsed. Each line must meet one of the following formats:

PARAM <name> <value>

MODELPARAM <name> <value>

DEVPARAM <name> <value>

DEVMODELPARAM <name> <value>

DEVPARAMs and DEVMODELPARAMs are parameters that are to appear at the DUT level. The difference between DEVPARAMs and DEVMODELPARAMs are that

DEVMODELPARAMs appear in model cards.

MODELPARAMs are PARAMs that appear in model cards. PARAMs and MODELPARAMs appear in the model parameters page.

<name> is the name of the model. <value> is its default value.
For certain saber parameters that can be altered, you may prepend SPECIAL to any of the PARAM keywords.

NODE <nodename>

Each NODE line declares a node to be recognized in IC-CAP setups. The order of the NODE lines must match the order the nodes are to appear when IC-CAP instantiates the instance card.

CIRCUITTYPE <x>

Here <x> is a single character. See above discussion of circuit types for proper values.

DEVICETYPE <x>

Here <x> is the name of the device for a device type circuit. See discussion about circuit types. This line is only required for saber.

UNRESOLVED <x>

Here <x> is the name of a model which was referenced in the netlist, but had no associated model card. In this case IC-CAP will try to find a model in its loaded list of models to insert.

ERROR: <x>

Here <x> is any arbitrary error message. The space after the colon is required. The entire line, including ERROR: will be reported in an error dialog.

DECKCOMPLETE

This should be the last line issued indicating that the parse is successful and that the output file is generated.

Translation Module Example

An example translation module, *$ICCAP_ROOT/src/mysim.c,* is provided with IC-CAP. The executable version of this program is *$ICCAP_ROOT/bin/mysim.* The following line is an example for adding a simulator called *mysim* to the IC-CAP simulator list:

```
mysim spice2 $ICCAP_ROOT/bin/mysim "" CAN_PIPE
```

where:

The simulator *mysim* uses *spice2* as the template simulator.
mysim is a user-written module that does the following:

- Translates a *spice2* input format deck to a *mysim* input format deck.
- Makes the call to the user's simulator. In this example, the executable simulator is *spice2.*
- Translates the user's binary output format to *spice2* binary output format.
- Optionally sends information to the output text file.

mysim is located in the *$ICCAP_ROOT/bin* directory.

The current host computer can perform a *mysim* simulation. The quotation marks ("") mean that no remote host is specified and therefore the simulation can be done on the current host machine.
The simulator *mysim* is capable of piped simulations.

After creating a translation module, you must compile it, using the system command:

```
cc -o mysim mysim.o -lm
```

Note
Whenever *$ICCAP_ROOT/iccap/lib/usersimulators* is modified, always restart IC-CAP to read the new simulator configuration. This file may be a symbolic link on SunOS so that each host served by a single file server can have a different simulator configuration.

Reserved Simulator Names

The following simulator names are reserved by IC-CAP and you cannot assign the same name to a different simulator:

spice2, spice3, hpspice

The following simulator names are defined in the *usersimulators* file but you can change their name and assign the same name to a different simulator.

hspice, saber, eldo, precise, spectre, spectre_spi, pspice, hpeesofsim

Simulator Argument Syntax

The command syntax for each simulator differs depending on whether a piped or non-piped simulation is being invoked. For details, refer to the documentation for each simulator (*ADS Simulator* (simulation), *Saber Simulator* (simulation), *SPECTRE Simulator* (simulation), *SPICE Simulators* (simulation)).

Remote Simulation

You can perform a simulation on a computer other than your computer by using the remote simulation feature. You can perform remote simulation due to any of the following reasons:

- Running the simulation on a faster machine
- Running the simulation on a computer authorized to run a particular simulator
- Running the simulation with a simulator that is not supported on the machine running IC-CAP, but supported on a remote machine

Before you Begin with Remote Simulation

To execute a remote simulation, the remote machine must meet the following requirements:

- The remote machine must possess Linux, SunOS, or a similar operating system that supports execution of Berkeley's remote shell *(remsh* or *rsh)* and remote file copy *(rcp)* commands. Alternatively, if the machine supports secure shell *(ssh)* and secure shell copy *(scp)*, IC-CAP can be configured to this mechanism by modifying `iccap.cfg` file.

  Note
  The remote machine must be set to receive secure shell connections. The machine running IC-CAP must have `ssh` and `scp` installed for this mechanism to function.

- Both local and remote machines are *familiar* with each other. This means that both the machines are connected through a network and the IP address database is updated to respond to the other machine. This database is usually found in */etc/hosts.*
- Allows remote shell and copy program execution from your local host without entering a password (relaxed network security). With ssh/scp, this requires generation of an authentication key. A utility is provided at *$ICCAP_ROOT/examples/model_files/misc/Setup_SSH.mdl* to assist with the generation of the key.
- Allows for the removal of files using */bin/rm*.
- The remote machine must contain */var/tmp* directory to write temporary files unless an alternate directory is specified in the *usersimulators* file for that simulators *host_name* field. See *host_name* (simulation) for more information.

The procedure for setting up the appropriate network security for your simulator depends on the remote host operating system.

Remote Simulation Algorithm

The name of the remote host is specified in the `usersimulators` file under *$ICCAP_ROOT/iccap/lib* file path. Remote simulation is supported in both CAN_PIPE and CANNOT_PIPE mode for *most* simulators. However, few simulators may only work in CANNOT_PIPE mode. See *Linking a Simulator to IC-CAP* (simulation).

The machine name for a simulator in the `usersimulators` file determines where each simulator runs.

- When a remote machine is not specified, the simulation occurs locally on your host computer.
- If the remote machine is specified, ensure that remote machine name is same as the current host name. When the remote machine is identical to the current host, the simulation is executed on the current host directly.
- If a remote machine is specified and remote machine name is not the same as the current host name, a remote simulation is performed by a remote shell command as shown below:

  For SunOS and Linux
  `/usr/ucb/rsh`

  On a PC
  `cygwin ssh`
  cygwin rsh can be configured in `iccap.cfg` file but for Windows XP only. The *cygwin rsh* shipped with IC-CAP does not function under Windows Vista or Windows 7.

- In non-piped simulation, necessary files are copied to the remote machine using a remote file copy command as shown below:

  For SunOS and Linux
  `/usr/ucb/rcp`

  On a PC
  `cygwin scp`
  cygwin rcp can be configured in iccap.cfg for Windows XP only. The *cygwin rcp* shipped with IC-CAP does not function with Windows Vista or Windows 7.

Network Security

When the remote commands (listed in the Remote Simulation Algorithm are executed, the current user ID is used to establish access to the remote machine. Therefore, it is necessary to have the same user ID on both local and remote machines. Also, the following files must be modified to allow remote program execution from a particular host.

For rsh/rcp

- */usr/adm/inetd.sec*
- */etc/hosts.equiv*
- . *rhosts*

For ssh/scp

- . *ssh/known_hosts*
- . *ssh/authorized_keys*

Reconfiguring IC-CAP to use ssh/scp or rsh/rcp

IC-CAP defaults to `rsh/rcp` on Linux or Sun and to `ssh/scp` on Windows operating system. Due to site security restrictions, it could be necessary to reconfigure IC-CAP on Linux or SunOS to use `ssh/scp`, or if you are using Windows XP, it could be appropriate to use `rsh/rcp`. To change the mechanism used by IC-CAP, you must modify the `C_REMOTE_SH_CMD` and `IC_REMOTE_CP_CMD` variables in your `iccap.cfg` file (for more details on configuring the `iccap.cfg` file, refer to *Customization and Configuration* (customization)).

The default values of `C_REMOTE_SH_CMD` and `IC_REMOTE_CP_CMD` variables are:

On Windows

```
IC_REMOTE_SH_CMD=%ICCAP_PC_UNIX_CMDS\ssh.exe
IC_REMOTE_CP_CMD=%ICCAP_PC_UNIX_CMDS\scp.exe
```

To change the default on Windows, use the following commands:

```
IC_REMOTE_SH_CMD=%ICCAP_PC_UNIX_CMDS\rsh.exe
IC_REMOTE_CP_CMD=%ICCAP_PC_UNIX_CMDS\rcp.exe
```

On Linux

```
IC_REMOTE_SH_CMD=/usr/bin/rsh
IC_REMOTE_CP_CMD=/usr/bin/rcp
```

On SunOS

```
IC_REMOTE_SH_CMD=/usr/ucb/rsh
IC_REMOTE_CP_CMD=/usr/ucb/rcp
```

To change the default on Linux or SunOS, specify the full path to ssh and scp depending on your system installation.

Validating rsh/rcp

When the security is set up, ensure that the following command returns the current date without any errors (substitute your remote machine name where < *remote_machine* > appears in the example).

On SunOS or Linux, type:

```
rsh <remote_machine> date
```

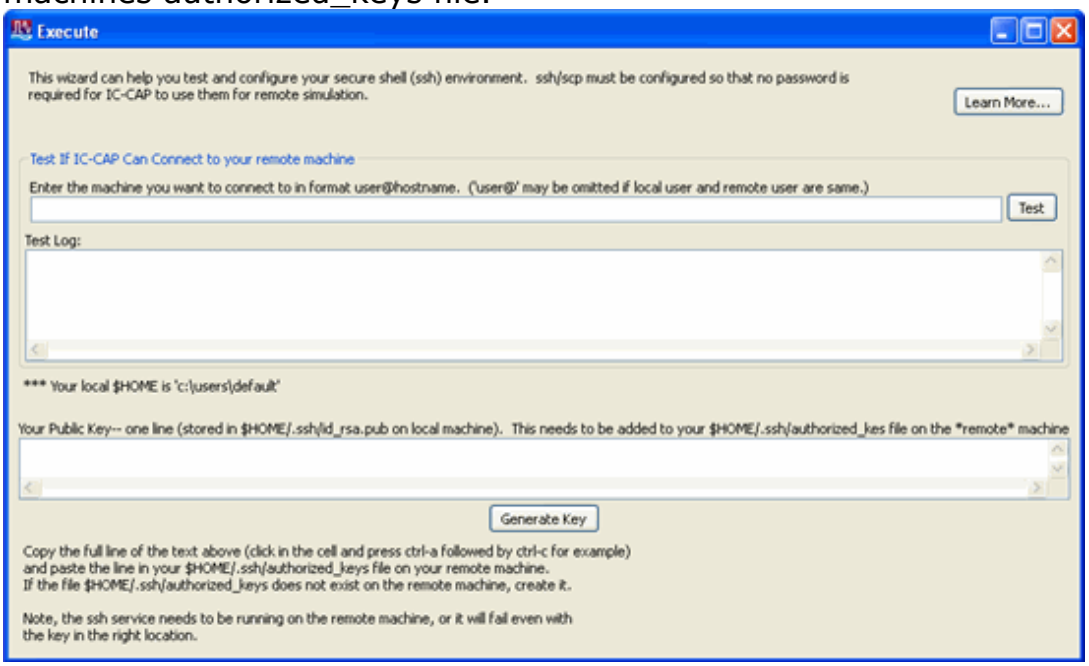
For the PC, execute the following line of PEL in an IC-CAP macro. The results are displayed in the IC-CAP Status window.

```
print system$("rsh <remote machine> date")
```

Validating ssh/scp

Perform the following steps to test remote simulation:

1. Open the `Setup_SSH.mdl` example file from the *examples/model_files/misc* directory. The `ssh_Config` model file icon is displayed in the IC-CAP Main window.
2. Double-click the model icon to open the remote test wizard. The remote test wizard tests the `ssh` connection and and generates a key which can be copied to the remote machines authorized_keys file.



> **Note**
> To use the remote test wizard, `ssh` and `scp` must be installed on the local machine.

This the recommended way to test remote simulation for Windows Vista and Windows 7.

Specifying the Remote simulator

If your remote simulator requires licensing variables, write a small shell-script on the remote host machine. The shell-script sets all the required environment variables before invoking the simulator. The shell script is specified in your *usersimulators* file instead of the actual simulator executable.

Example:

```
#!/bin/sh
LM_LICENSE_FILE=my_license_file.lic
export LM_LICENSE_FILE
PATH=/path/to/my/simulator/bin:$PATH
export PATH
/path/to/my/simulator/bin/xxxxx $*
```

The $* on the last line is required to pass along the IC-CAP command-line parameters.

Executing a Remote Simulation

After the *usersimulators* file is set up correctly and the network security is adjusted, the

steps for performing a remote simulation are identical to those required to perform a non-remote simulation. For more information, refer to *Performing a Simulation* (simulation).

Note
Ensure that there are no commands in the *.cshrc* file on the remote host that may generate output. Also, do not perform terminal related operations in your *.cshrc* file such as termset or stty. Since there is no physical terminal with remote shell commands, commands expecting one in your *.cshrc* file lead to errors.

Remote Simulation Examples

The following example specifications for running the template simulators remotely can be added to the *usersimulators* file in the *$ICCAP_ROOT/iccap/lib* filepath.

Note
If you prefer to preserve the defaults as shipped, you can add the specifications to your local, or Home, directory by copying and editing the usersimulators file. If you set the specifications in your home directory, you must change the pointers in your configuration file. Copy the file *$ICCAP_ROOT/config/iccap.cfg* to *$HOME/hpeesof/config/iccap.cfg*. See *Customization and Configuration* (customization) for additional information.

The examples contain sample user-assigned simulator names, remote host machine names, and directory path (on the remote machine) information. This information must be replaced by the actual names in your system. The purpose of these examples is to show the names of the simulator executable files.

You must specify a full path name for each simulator because the PATH variable on the remote machine may not have the necessary search path to find your simulator.

Note
User-assigned simulator names can be any name you choose except for the reserved names. See *Reserved Simulator Names* (simulation). To use a user-assigned simulator, ensure that the simulator name is listed in the first column of the *usersimulators* file, and then set your simulator in IC-CAP to the same name.

Running spectre on the Remote Solaris Machine

To run spectre on the remote Solaris machine called *cadencebox*, enter the following command:

```
remspectre_SS spectre443 /cadence/5.0.0/tools/bin/spectre "cadencebox"
CANNOT_PIPE
```

where,

spectre443 is the template for spectre version 4.4.3 and greater, which causes IC-CAP to parse its circuit page expecting spice syntax.

Running spectre from a PC

From a PC, enter the following command:

```
remspectre_SS spectre443.be /cadence/5.0.0/tools/bin/spectre "cadencebox"
CANNOT_PIPE
```

Since the Solaris machine's byte order is big endian, append the *.be* extension to the template name.

Running Native spectre on the remote Solaris Machine

To run native spectre on the remote Solaris machine called *cadencebox*, enter the following command:

```
remspectre spectre /cadence/5.0.0/tools/bin/spectre "cadencebox" CANNOT_PIPE
```

where,

spectre is the template for native spectre, which causes IC-CAP to parse its circuit page expecting native spectre syntax. See *SPECTRE Interfaces* (simulation).

Running Native spectre from a PC

From a PC, enter the following command:

```
remspectre spectre.be /cadence/5.0.0/tools/bin/spectre "cadencebox" CANNOT_PIPE
```

Since the Solaris machine's byte order is big endian, append the *.be* extension to the template name.

Running UCB SPICE 2G.6 on a Remote Machine

To run UCB SPICE 2G.6 on the remote machine called *spice2mach,* enter the following command:

```
spice2rem spice2 /usr/iccap/bin/ucbspice2g6 "spice2mach" CAN_PIPE
```

where:,

spice2rem represents the name of the simulator and *pipe* creates an interprocess channel that responds to read/write calls.

To run UCB SPICE 3E2 on the remote machine called *spice3mach,* enter the following command:

```
spice3rem spice3 /usr/iccap/bin/spice3e2 "spice3mach" CAN_PIPE
```

where,

spice3rem represents the name of the simulator and *pipe* creates an interprocess channel that responds to read/write calls.

Running HPSPICE on a Remote Machine

To run HPSPICE on the remote machine called *hpspicemach,* enter the following command:

```
hpspicerem hpspice /usr/iccap/bin/shpspice "hpspicemach" CAN_PIPE
```

where,

hpspicerem represents the name of the simulator and *pipe* creates an interprocess channel that responds to read/write calls.

When performing remote simulations using the HPSPICE simulator, *both* of the executable files called shpspice and spice2.4n1 must be present on the remote machine.

To run HSPICE on the remote machine called *hspicemach* , enter the following command:

```
hspicerem hspice /usr/bin/hspice "hspicemach" CANNOT_PIPE
```

where,

hpspicerem represents the name of the simulator and *pipe* creates an interprocess channel that responds to read/write calls

Depending on the version of HSPICE you have installed, the execution script called *hspice* can exist in a different directory path from /usr/bin/hspice. In this case, create a symbolic link from /usr/bin/hspice to the actual *hspice* script that will be called. For example, if your hspice script exists under /usr/meta/h9007/bin/hspice, then execute the following command to create the required symbolic link:

```
ln -s /usr/meta/h9007/bin/hspice /usr/bin/hspice
```

Note
You must be in the root directory when executing the above command.

Running ELDO on a Remote Machine

To run ELDO on the remote machine called *eldomach* ,enter the following command:

```
eldorem eldo<anacad_root>/eldo/<version>/com/eldo"eldomach" CANNOT_PIPE
```

where,

<anacad_root> and *<version>* are replaced with the home directory of the ANACAD software and the current version number of ELDO, respectively.
eldorem represents the name of the simulator.
pipe creates an interprocess channel that responds to read/write calls.

Running Saber on a Remote Machine

To run Saber on the remote machine called *sabermach*, enter the following command:

```
saberrem saber /usr/saber/bin/saber "sabermach" CAN_PIPE
```

where,

saberrem represents the name of the simulator and *pipe* creates an interprocess channel

that responds to read/write calls.

Running HPEESOFSIM on a Remote Machine

To run HPEESOFSIM on the remote machine called *hpsimmach*, enter the following command:

```
hpsimrem hpeesofsim <simulator path> "hpsimmach" CAN_PIPE
```

where,

hpsimrem represents the name of the simulator and *pipe* creates an interprocess channel that responds to read/write calls.
hpeesofsim is the simulator name.
The third field is the path to the simulators location installed on the remote machine.

To launch an external simulation, copy the file *$ICCAP_ROOT/bin/hpeesofsim_start* to some location on the remote machine. Modify the file (as explained within the file) to set HPEESOF_DIR and HPEESOFSIMFRONT_DIR for the remote machine. Then, ensure that *hpeesofsim_front* is on the remote machine. If the remote machine contains the same OS as the local machine, then you can copy *$ICCAP_ROOT/bin/hpeesofsim_front* from the local machine. If the remote machine has a different architecture, *hpeesofsim_front* for all architectures can be found on your distribution CD under the subdirectory *simlinks*. Finally, modify the file *usersimulators* to refer to the remote host and the path to *hpeesofsim_start* on the remote host.

Simulators in IC-CAP

- *ADS Simulator* (simulation)
- *SPECTRE Simulator* (simulation)
- *HSPICE Simulator* (simulation)
- *SPICE Simulators* (simulation)
- *ELDO Simulator* (simulation)
- *Saber Simulator* (simulation)

ADS Simulator

About ADS Simulator in IC-CAP

This section describes the details of using the Advanced Design System (ADS) Simulator with IC-CAP. For general information on IC-CAP simulation, refer to *Simulation* (simulation).

Note
The PC version of IC-CAP supports ADS version 2002 or newer. Older versions of ADS can not be used with the PC version of IC-CAP.

IC-CAP supports the following ADS features:

- DC, Small Signal AC, Small Signal S-Parameter, and Transient analysis options
- Parameter sweeps for device and circuit simulation
- Temperature sweeps
- Hierarchical simulation
- Variables
- Constants
- Expressions
- Spectre circuit page
- Spice circuit page

The ADS Optimizer features are not currently supported in IC-CAP. IC-CAP optimization (different from the ADS Optimizer) of simulated data to target data is supported.

The ADS simulator supports the following analysis types:

- DC
- AC
- 2-port
- Multiport
- Transient
- Noise
- Capacitance Voltage (CV)
- Time-Domain Reflectometry (TDR)
- Steady State Harmonic Balance

Note
2-port simulation with high frequency noise is supported to extract noise parameters such as noise figure, optimum source reflection coefficients, equivalent noise resistance data, minimum noise figure data, and equivalent noise temperature data.

IC-CAP does not add extra circuitry in order to perform a 2-port simulation since this is a standard type in ADS.

ADS Interfaces

IC-CAP provides three template names to interface to the ADS simulator's Circuit and Test Circuit pages:

- *hpeesofsim* uses native ADS simulator syntax
- *spmodeads* uses spectre simulator syntax, and
- *hspicemodeads* uses hspice simulator syntax

All interfaces use native ADS simulator syntax to specify the sweep and output requests.

`usersimulators` file should have a line similar to the following:

- To specify *hpeesofsim*, type the following:

```
hpeesofsim hpeesofsim $ADS_DIR/bin/iccapinterface "" CAN_PIPE
```

To specify *spmodeads*, type the following:

```
spmodeads spmodeads $ADS_DIR/bin/iccapinterface "" CAN_PIPE
```

To specify *hspicemodeads*, type the following:

```
hspicemodeads hspicemodeads $ADS_DIR/bin/iccapinterface "" CAN_PIPE
```

The first field can be any name as per your choice and it will show up in your simulator list. It can be used with the *SIMULATOR* variable.

The *hpeesofsim* , *spmodeads* and *hspicemodeads* lines shown above are in the `usersimulators` file by default.

When using the *spmodeads* interface, refer to *Circuit Model Descriptions* (simulation) in *SPECTRE Simulator* (simulation) for spectre syntax for the Circuit and Test Circuit pages. When using the *hspicemodeads* interface, refer to *Circuit Model Descriptions* (simulation) in *SPICE Simulators* (simulation) for spice syntax for the Circuit and Test Circuit pages.

System Requirements

Hardware and Operating System Requirements

The ADS Simulator on IC-CAP is supported on the following platforms:

- Linux RedHat Enterprise 4.0 or Linux Novell SUSE SLES 9
- Solaris 10
- Microsoft Windows XP or Microsoft Windows Vista.

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.

Setting Environment Variables

Before running the ADS Simulator, set the environment variable *HPEESOF_DIR* on UNIX or *ADS_DIR* on Windows to point to the ADS Simulator's installation location.

- To set *HPEESOF_DIR* using the Korn Shell, add the following to your *~/.profile*.

```
export HPEESOF_DIR=< ADS_install_directory >
```

- To set *HPEESOF_DIR* using the C Shell, add the following to your *~/.cshrc*.

```
setenv HPEESOF_DIR < ADS_install_directory >
```

- To set *ADS_DIR* for Windows 2000, right click on **My Computer** and select **Properties**. Click on the **Advanced** tab. Then select **Environment Variables** and set *ADS_DIR* either for the local user or system wide, depending on your needs.
- To set *ADS_DIR* for Windows NT 4.0, right click on **My Computer** and select **Properties**. Click on the **Environment** tab. Then set *ADS_DIR* either for the local user or system wide, depending on your needs. You may need to log off and log back onto the computer for the new variable to be found by IC-CAP.

ADS Simulation Example

The circuit description is predefined for all IC-CAP configuration files. Enter this description if a new model is being defined; edit the description to fit specific needs. The syntax is identical to the syntax used for describing circuits in a typical ADS simulation deck.

This ADS simulation example will use the IC-CAP supplied Model hpsimnpn.mdl.

1. Choose **File > Examples > model_files/bjt/hpsimnpn.mdl**. Choose **OK**.
2. View the description by clicking the **Circuit** tab.
   The circuit description is shown in the following figure. This deck describes the circuit (in this case, a single device) to be used in the simulation.

ADS Circuit Description Deck for an NPN Bipolar Transistor

```
; Simulation Input File for BJT
options ascii=no
model npnbjt BJT NPN=yes \
Is=401.5a Bf = 87.01 \
Nf = 995.5m Vaf = 84.56 \
Ikf = 11.95m Ise = 34.05f \
Ne = 1.594 Br = 10.79 \
```

```
Nr = 1.002 Var = 9.759 \
Ikr = 23.7m Isc = 1.095f \
Nc = 1.100 Rb = 9.117 \
Irb = 1.613m Rbm = 5.620 \
Re = 1.385 Rc = 9.292 \
Xtb = 1.7 Eg = 1.110 \
Xti = 3.000 Cje = 1.312p \
Vje = 1.110m Mje = 347.5m \
Tf = 52.74p Xtf = 5.625 \
Vtf = 2.678 Itf = 23.82 \
Ptf = 154.1 Cjc = 1.396p \
Vjc = 451.1m Mjc = 192.4m \
Xcjc = 300m Tr = 1.00n \
Cjs = 99.85f Vjs = 813.7m \
Mjs = 350.9m Fc = 500.0m \
Tnom = 27
npnbjt:Q1 C B E S
```

To view the input and output for the fearly setup, click the **DUTs-Setups** tab and select **fearly**.
The Measure/Simulate folder appears with the inputs vb, vc, ve, and vs, and the output ic. The vc input specifies a voltage source at node C that sweeps linearly from 0 to 5V in 21 steps. The ic output specifies that current at node C be monitored.
In the Plots folder, icvsvc is specified so that the results of the simulation can be viewed graphically.
To simulate, click the **Simulate** button in the Measure/Simulate folder. The Status line displays **Simulate in progress**. Under most configurations, the ADS status window will appear. For more information about these configurations, see *Piped ADS Simulations* (simulation).
When the simulation is complete, the Status line displays **Simulate Complete**.
To view the results of the simulation, right-click on **fearly**, then choose **Plots > icvsvc**. (This is a shortcut for displaying the plot from the *Plots* folder.) The plot displays measured data represented by solid lines and simulated data represented by dashed lines.

Note
For syntax examples of running a remote simulation, refer to *Remote Simulation Examples* (simulation).

The Simulation Debugger

When using ADS with the Simulation Debugger to perform an IC-CAP simulation (as opposed to a manual simulation), an output text file consists of only the computational analysis information. An example of a typical AC analysis output text file is as follows:

```
HPEESOFSIM (ver. 03.00 -- 12/14/01 09:28:45)
Copyright Agilent Technologies, 2004
CT ct1[1] </var/tmp/ICCAAa18727>   VBGROUND.Vdc=(700 mV->720 mV)
DC ct1[1].dc1[1/3] </var/tmp/ICCAAa18727>  VBGROUND.Vdc=700 mV  VCGROUND.Vdc=(0 V->5 V)
DC ct1[1].dc1[2/3] </var/tmp/ICCAAa18727>  VBGROUND.Vdc=710 mV  VCGROUND.Vdc=(0 V->5 V)
DC ct1[1].dc1[3/3] </var/tmp/ICCAAa18727>  VBGROUND.Vdc=720 mV  VCGROUND.Vdc=(0 V->5 V)
-------------------
Simulation finished.
-------------------
```

This file does not include the resulting data. To generate a more informative output text file, change the *ASCII_Rawfile* option in the Input File from *ASCII_Rawfile=no* to

ASCII_Rawfile=yes and perform a manual simulation. An output text file that includes the simulated output data values is produced. The *ASCII_Rawfile* option is set to *no* by IC-CAP before every simulation so that the binary raw data file is generated by ADS. IC-CAP needs the binary raw data file to read the resulting simulation data. However, this data is not needed for a manual simulation.

ADS version 1.3 requires that the option *UseNutmegFormat* be set to *yes* to cause ADS to generate the binary raw data file required by IC-CAP. If the *UseNutmegFormat* option is not specified, the default is *UseNutmegFormat = yes.* If you set *UseNutmegFormat = no,* ADS will generate an output data format that IC-CAP cannot understand.

Piped and Non-Piped ADS Simulations

Piped ADS Simulations

Specifying CAN_PIPE (the default) in your *usersimulators* file for the ADS simulator enables IC-CAP to take advantage of the tune mode built into the ADS simulator. This mode permits changing parameters of a simulation without requiring the simulator to be relaunched. This greatly reduces the time required for optimizations to run. However, each setup requires a new simulator to be launched. By default, IC-CAP permits up to 3 ADS simulators to be running at once so that an optimization across as many as 3 setups can be completed in the fastest time possible. Certain large simulations may require a great deal of system resources and having 3 simulations currently active can degrade system performance. If you encounter this problem, you can set the MAX_PARALLEL_SIMULATORS system variable to 1 or 2. If your system can handle more than 3 simulators in parallel and you need to optimize across more than 3 setups at a time, the value of MAX_PARALLEL_SIMULATORS can be increased.

When CAN_PIPE mode is used, the ADS simulator will bring up a status window during simulation. The first time the simulator is launched it can take several seconds for this window to appear. Once it is open, successive simulations will attach to the same status window. Each time a new setup is simulated, a new simulator must be started. There is a certain start-up delay associated with each invocation. This will be much shorter than the very first invocation which needed to launch the status window. Successive simulations of a setup which has been previously simulated will return in the shortest time as the simulator does not need to be reinvoked.

Opening the Simulation Debugger will terminate all running simulators, and close the ADS status window. Simulations done with the Simulation Debugger window open are performed in non-piped mode and thus the ADS status window is not opened.

In situations when you want to use the *$mpar* or *$dpar* feature in *#echo* lines for ADS netlists, you *must* enter names properly. The proper ADS name syntax is a dot-separated name, such as *NPN.Bf*. If you fail to use a proper name, simulations will yield incorrect results when you try to use the simulator in CAN_PIPE mode. If names cannot be revised, use CANNOT_PIPE.

This was especially problematic for userdefined models requiring many *#echo* lines using the *$mpar* feature in order for IC-CAP to parse it properly. This problem occurs when the technique used to implement userdefined models in ADS is declaring 2 new components, 1

a modelform and another an instance. This implementation of user-defined models led to the requirement for *#echo* lines. The modelform component looked like any other ADS netlist component, but it had no nodes. The parser is modified for IC-CAP 2001 to recognize a nodeless component as a userdefined model; however, only in the context of a subcircuit. If you want to create *this* type of userdefined model in ADS, then you *must* use a subcircuit. Doing so eliminates the need for *#echo* lines in the netlist and the subcircuit will parse and simulate properly.

Non-Piped ADS Simulations

Execute a simulation with the Simulation Debugger ON to perform a non-piped simulation. ADS is capable of performing piped simulations, which enables you to turn the Simulation Debugger OFF without requiring that ADS be restarted for every simulation.

Circuit Model Descriptions

This section explains the circuit descriptions for the ADS simulator.

Selecting Simulator Options

ADS simulation options are specified using the HPEESOFSIM_OPTIONS variable in the Setup DUT or System Variable tables. Enter the options in the value section of the variable exactly as they should appear in the ADS options command.

Entering Circuit Descriptions

The circuit description is entered into the IC-CAP Circuit Editor or the Test Circuit Editor. The circuit description includes the necessary definitions of devices, sources and components, as well as node connections and model descriptions. ADS accepts a netlist description that is different from SPICE and Saber simulators.

Note
IC-CAP accepts a modified form of a netlist that enables you to use binning. To simulate a netlist with binned models from IC-CAP, you must declare the bin model (and only the bin model) immediately following the subcircuit definition. You must declare each *Model[x]*= to be a name of the form *XCKT.modname* since that is how IC-CAP netlists the bin model.

Parameter Table Generation

The circuit description is parsed by IC-CAP and specific model information (such as parameters and their corresponding values) as well as circuit component values are reflected in the Parameters table. Model parameters and component values specified in the circuit description entered in the Circuit Editor are saved in the Parameters table. Device parameters specified in the model call statement are saved in the DUT Parameters table-unless a Test Circuit is specified, in which case, parameter values specified in the test circuit description are saved in the DUT Parameters table.

By default, all parameter names will be converted to uppercase, since most extraction routines look for parameters named with all uppercase letters. Some extraction routines (e.g., *Root* models and *EExxx* models) require all lowercase letters. In these *.mdl* files, the variable HPEESOFSIM_USE_LOWER_CASE_PARAMS is declared to override the default behavior. If you want to write extraction routines using the native mixed case parameters,

declare the variable HPEESOFSIM_USE_MIXED_CASE_PARAMS in your model file. For a description of these functions, see the System Variables.

Non-numeric Parameter Values

ADS allows non-numeric values for a number of parameters in predefined component definitions. One example is the BJT model parameter *npn*. This parameter can take on the value of *yes* if it is an nmos device. Alpha format parameters do not appear in the IC-CAP Parameters table but do appear in the simulation input decks.

Circuit descriptions must be entered with valid model and parameter names for the particular model being used.

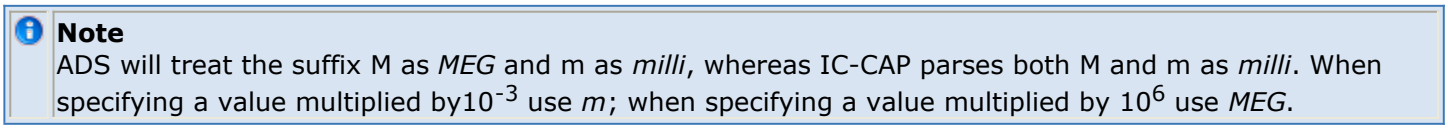
Node Names

ADS accepts alphanumeric names as well as numbers to represent nodes. There is no limit on the number of characters allowed in a node name; however, delimiters or non-alphanumeric characters are not allowed. Also, a node name that begins with a digit must consist only of digits.

Numeric node names are discouraged as they will result in warnings during simulation that the results will not be displayed properly in ADS Data Display Server (DDS). However, these warnings are of no consequence to ICCap.

Comments

To indicate comments in an ADS input deck, start an input line with a semicolon (;). All text on the line following the semicolon will be ignored.



Note
ADS will treat the suffix M as *MEG* and m as *milli*, whereas IC-CAP parses both M and m as *milli*. When specifying a value multiplied by10^{-3} use *m*; when specifying a value multiplied by 10^{6} use *MEG*.

Device Model Descriptions

A device model is used to characterize a single ADS-defined element of any type. This specification requires a model definition that describes the device and an instance statement that calls the model definition.

The model description specifies the value of a device model that describes a particular element. When a parameter is not specified, the default value in the model is used. The general form of the model definition is:

```
model MNAME TYPE PNAME1 = PVAL1 PNAME2 = PVAL2...
```

where

MNAME is the model name. (Regardless of the model name entered into the MNAME field of the ADS model definition statement, IC-CAP substitutes this field with the name of the Model as it is called in the Main window when the simulator

input deck is built.)

TYPE is a valid ADS element type.

PNAMEs are parameter names available for the particular model type.

PVALs are the parameter values.

A backslash immediately followed by a return (no space between the backslash and the return) at the end of a line indicates that the statement is continued on the next line This continuation character is often used for easier readability when specifying the model description.

The general form of the instance statement that calls the device model is:

```
TYPE :DNAME NNAME1 NNAME2...NNAMEN DPAR1 = DVAL1
DPAR2 = DVAL2...DPARN = DVALN
```

where

TYPE is the instance type descriptor. This field can contain either the ADS instance type name or a user-supplied model or subcircuit name.

DNAME is the device name.

NNAMEs denote node names.

DPAR is a predefined DUT parameter name.

DVAL is the specified DUT parameter value. Refer to *General Syntax* (simulation) for DUT parameter names available for each model.

Subcircuit Model Descriptions

A subcircuit definition represents a circuit that contains more than 1 device. The syntax for defining a subcircuit is identical to the syntax used for the ADS input language.

The general form of the subcircuit definition is:

```
define SUBCKTNAME (NNAME1 NNAME2 ...NNAMEN)
parameters PAR1 = VAL1 PAR2 = VAL2 ...PARN = VALN
< body of subcircuit >
end SUBCKTNAME
```

where

SUBCKTNAME is the name of the subcircuit.

NNAMEs are the node names of the external nodes of the subcircuit. These external nodes are used to connect the subcircuit to another circuit.

PARs are the names of the parameters passed into the subcircuit. These parameters are optional in a subcircuit definition.

If parameters are specified, the assigned default values VAL are also optional. A parameter is assigned to this default value if the parameter is not specified in the subcircuit call.

The body of the subcircuit contains element statements. It can contain calls to other subcircuits but it cannot contain other subcircuit definitions.

The subcircuit definition is completed using the end SUBCKTNAME statement.

Calling a subcircuit definition allows you to insert all instances specified within the subcircuit into the circuit. The call requires a syntax identical to the syntax used in the ADS input language for any instance statement. The general form of the instance statement is:

```
TYPE :INAME NNAME1 NNAME2....NNAMEN PAR1 = VAL1 PAR2 =
VAL2......PARN = VALN
```

(While the syntax shown here is correct, passed parameters are ignored by IC-CAP.)

where

TYPE is the instance type descriptor. If a subcircuit is being called, this field would contain the subcircuit name denoted by SUBCKTNAME.

INAME is the instantiated name of the subcircuit.

NNAMEs denote node names.

PARs are the subcircuit parameter names.

VALs are the specified subcircuit parameter values.

The following is an example of a complete subcircuit definition and subcircuit call.

- Added by IC-CAP for output format/etc.

```
Options ASCII_Rawfile=no UseNutmegFormat=yes
```

- Defined by the user in the Circuit folder:

```
;Simulation Input File in hpeesofsim Input Deck Format
global RC1_r=4352
define hpsimopamp (2 3 4 6 7 )
; Internal OpAmp circuit
; using Boyle-Pederson Macro Model
; Input differential amplifier
NPN1:Q1  10 2 12
NPN2:Q2  11 3 13
model NPN1 BJT NPN=yes  \
```

```
Is = 8E-16  \
Bf = 52.81
model NPN2 BJT NPN=yes  \
Is = 8.093E-16  \
Bf = 52.66
R:RC1  7 10 R=RC1_r
R:RC2  7 11 R=4352
C:C1  10 11 C=4.529E-12
R:RE1  12 14 R=2392
R:RE2  13 14 R=2392
R:RE  14 0 R=7.27E+06
C:CE  14 0 C=7.5E-12
; Power dissipation modeling resistor
R:RP  7 4 R=1.515E+04
; 1st gain stage
#uselib "ckt", "VCCS"
VCCS:GCM  14 0 0 15 G=1.152E-09
VCCS:GA  10 11 15 0 G=0.0002298
R:R2  15 0 R=1E+05
; Compensation capacitor
C:C2  15 16 C=3E-11
; 2nd gain stage
VCCS:GB  15 0 16 0 G=37.1
R:RO2  16 0 R=489.2
DMOD1:D1  16 17
DMOD1:D2  17 16
model DMOD1 Diode  \
Is = 3.822E-32
R:RC  17 0 R=0.0001986
VCCS:GC  6 0 0 17 G=5034
; Output circuit
R:RO1  16 6 R=76.8
DMOD2:D3  6 18
DMOD2:D4  19 6
model DMOD2 Diode  \
Is = 3.822E-32
V_Source:VC  7 18 Vdc=1.604
V_Source:VE  19 4 Vdc=3.104
; Input diff amp bias source
I_Source:IEE  14 4 Idc=2.751E-05
end hpsimopamp
```

- Defined by the user in the Test Circuit folder:

```
; Inverting Amplifier
define inv_amp (1 2 3 4 6 7 )
hpsimopamp:X1  2 3 4 6 7
R:Rf  6 2 R=1E+04
R:Rin  2 1 R=2000
R:Rgnd  3 0 R=0.001
end inv_amp
```

- Added by IC-CAP to the circuit description:

```
inv_amp:XCKT n1 n2 n3 n4 n5 n6
; START SOURCES
V_Source:V1GROUND n1 0 Vdc=0 Vac=1
```

```
V_Source:V7GROUND n6 0 Vdc=15
V_Source:V4GROUND n4 0 Vdc=-15
; END SOURCES
R:RO2 n2 0 R=100MEG
R:RO3 n3 0 R=100MEG
R:RO5 n5 0 R=100MEG
SweepPlan:swpfreq  Start = 1000 Stop = 1e+07 Dec = 3
AC:ac1 SweepPlan=swpfreq SweepVar="freq"
```

ADS Parameter Sweeps

When using the ADS simulator in IC-CAP, the method of specifying parameter sweeps is unique when performing simulations with a defined Test Circuit at the DUT level which references a subcircuit style netlist from the model level.

Note
When performing parameter sweeps, the name of the parameter to be swept must be recognized by ADS, since the analysis is performed from within the simulator. If you need to sweep a parameter from the model level netlist in this scenario, you must define a variable manually-- both within the netlist and in the IC-CAP variable table and sweep the variable.

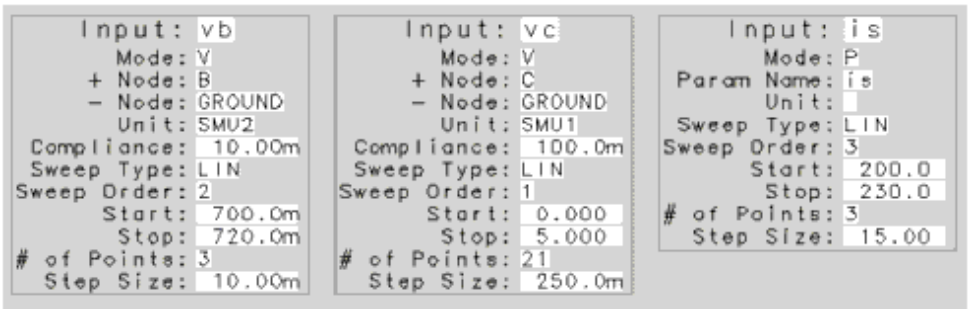
Normal Parameter Sweep Specification

To sweep a parameter in an ADS device simulation, or in a circuit simulation without a test circuit:

1. Add an input specification of mode P to the Setup. Enter the name of the parameter as it appears in the Parameters table.
2. Enter the sweep type and values.

The Device Simulation Parameter Sweep example uses the *hpsimnpn.mdl* model with an input of mode P to the *fearly* setup. This input specifies a linear sweep of the parameter from 200.0e-15 to 230.0e-15 amperes in steps of 15.0e-15 amperes.

ADS Device Simulation Parameter Sweep Setup Example



During the simulation, IC-CAP generates the following input deck.

```
Options\
ASCII_Rawfile=no UseNutmegFormat=yes
; Simulation Input File for BJT
model npn BJT NPN=yes  \
Is =  2.704E-16  \
```

```
Bf =  86.16  \
Nf =  0.979  \
Vaf =  86.95  \
Ikf =  0.01491  \
Ise =  1.886E-14  \
Ne =  1.522  \
Br =  8.799  \
Nr =  0.9967  \
Var =  9.757  \
Ikr =  0.02369  \
Isc =  1.095E-15  \
Nc =  1.1  \
Rb =  8.706  \
Irb =  0.001509  \
Rbm =  5.833  \
Re =  1.385  \
Rc =  10.68  \
Xtb =  0  \
Eg =  1.11  \
Xti =  3  \
Cje =  1.312E-12  \
Vje =  0.6151  \
Mje =  0.2052  \
Tf  =  4.781E-11  \
Xtf =  4.359  \
Vtf =  3.237  \
Itf =  0.01753  \
Ptf =  176.2  \
Cjc =  1.394E-12  \
Vjc =  0.5428  \
Mjc =  0.2254  \
Xcjc =  1  \
Tr  =  5.099E-09  \
Cjs =  1.004E-13  \
Vjs =  0.5668  \
Mjs =  0.2696  \
Fc =  0.5  \
Tnom =  27
npn:devckt 1 2 3 4
; START SOURCES
V_Source:VBGROUND 2 0 Vdc=0
V_Source:VCGROUND 1 0 Vdc=0
V_Source:VEGROUND 3 0 Vdc=0
V_Source:VSGROUND 4 0 Vdc=-3
; END SOURCES
SweepPlan:swp1 Start=0 Stop=5 Step=0.25
SweepPlan:swp2 Start=0.7 Stop=0.72 Step=0.01
DC:dc1 SweepPlan=swp1 SweepVar="VCGROUND.Vdc"
ParamSweep:ct1 SimInstanceName="dc1" SweepPlan=swp2
SweepVar="VBGROUND.Vdc"
```

Circuit Simulation Plus Test Circuit Parameter Sweep

Specifying a parameter sweep for a circuit simulation which includes a Test circuit requires a different approach from a parameter sweep for a device simulation.

To sweep a parameter at the model level in an ADS circuit simulation:

1. Specify a global variable in the ADS circuit description and set it to an initial value.

2. Set the value of the parameter in the circuit description equal to the global variable name.
3. Add a variable in IC-CAP with the same name as the global ADS parameter.
4. Add an input specification of mode P to the Setup.
5. Enter the global variable name in the Name field of the Input table.
6. Enter the sweep type and values.

DUT Level parameters can be specified in the normal way covered in the previous section.

Example Circuit Simulation Parameter Sweep

The Circuit Simulation Parameter Sweep example, uses a model for opamp simulation. The following line is included with the circuit description:

global RC1_r=4352

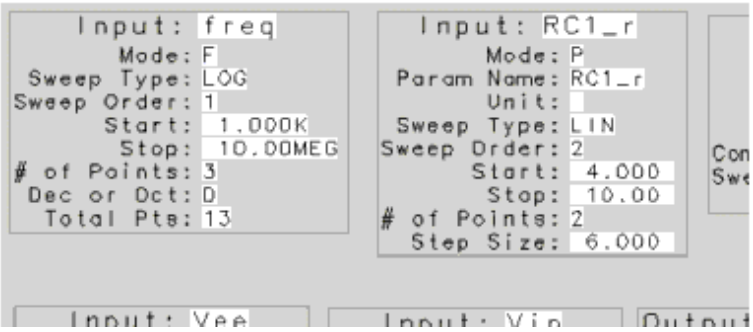
This complete circuit description is shown below.

```
;Simulation Input File in hpeesofsim Input Deck Format
global RC1_r=4352
define opamp1 (2 3 4 6 7)
; Internal OpAmp circuit
; using Boyle-Pederson Macro Model
; Input differential amplifier
NPN1:Q1 10 2 12
NPN2:Q2 11 3 13
model NPN1 BJT NPN=yes \
Is = 8E-16 \
Bf = 52.81
model NPN2 BJT NPN=yes \
Is = 8.093E-16 \
Bf = 52.66
R:RC1 7 10 R=RC1_r
R:RC2 7 11 R=4352
C:C1 10 11 C=4.529E-12
R:RE1 12 14 R=2392
R:RE2 13 14 R=2392
R:RE 14 0 R=7.27E+06
C:CE 14 0 C=7.5E-12
; Power dissipation modeling resistor
R:RP 7 4 R=1.515E+04
; 1st gain stage
#uselib "ckt", "VCCS"
VCCS:GCM 14 0 0 15 G=1.152E-09
VCCS:GA 10 11 15 0 G=0.0002298
R:R2 15 0 R=1E+05
; Compensation capacitor
C:C2 15 16 C=1E-11
; 2nd gain stage
VCCS:GB 15 0 16 0 G=37.1
R:RO2 16 0 R=489.2
DMOD1:D1 16 17
DMOD1:D2 17 16
model DMOD1 Diode \
Is = 3.822E-32
R:RC 17 0 R=0.0001986
```

```
VCCS:GC 6 0 0 17 G=5034
; Output circuit
R:RO1 16 6 R=76.8
DMOD2:D3 6 18
DMOD2:D4 19 6
model DMOD2 Diode \
Is = 3.822E-32
V_Source:VC 7 18 Vdc=1.604
V_Source:VE 19 4 Vdc=3.104
; Input diff amp bias source
I_Source:IEE 14 4 Idc=2.751E-05
end opamp1
define inv_amp (1 2 3 4 6 7 )
opamp1:X1  2 3 4 6 7
R:Rf  6 2 R=1E+004
R:Rin  2 1 R=2000
R:Rgnd  3 0 R=0.001
end inv_amp
inv_amp:XCKT n1 n2 n3 n4 n5 n6
; START SOURCES
V_Source:V1GROUND n1 0 Vdc=0 Vac=polar(1,0)
V_Source:V7GROUND n6 0 Vdc=15
V_Source:V4GROUND n4 0 Vdc=-15 ; END SOURCES
R:RO2 n2 0 R=100MEG
R:RO3 n3 0 R=100MEG
R:RO5 n5 0 R=100MEG
SweepPlan:swpfreq Start=1000 Stop=1e+07 Dec=3
SweepPlan:swp1 Start=4 Stop=10 Step=6
AC:ac1 SweepPlan=swpfreq SweepVar="freq"
ParamSweep:ct1 SimInstanceName="ac1"
SweepPlan=swp1 SweepVar="RC1_r"
```

In this example, the value of R:RC1 is set to *RC1_r*. You must also add a variable called RC1_r to the IC-CAP model variables table and set the variable to a value, such as, 4.000K. In the example model, you must then add an input call *RC1_r* to the setup. The Inputs table is shown in the following figure.

ADS Circuit Parameter Sweep Setup Example



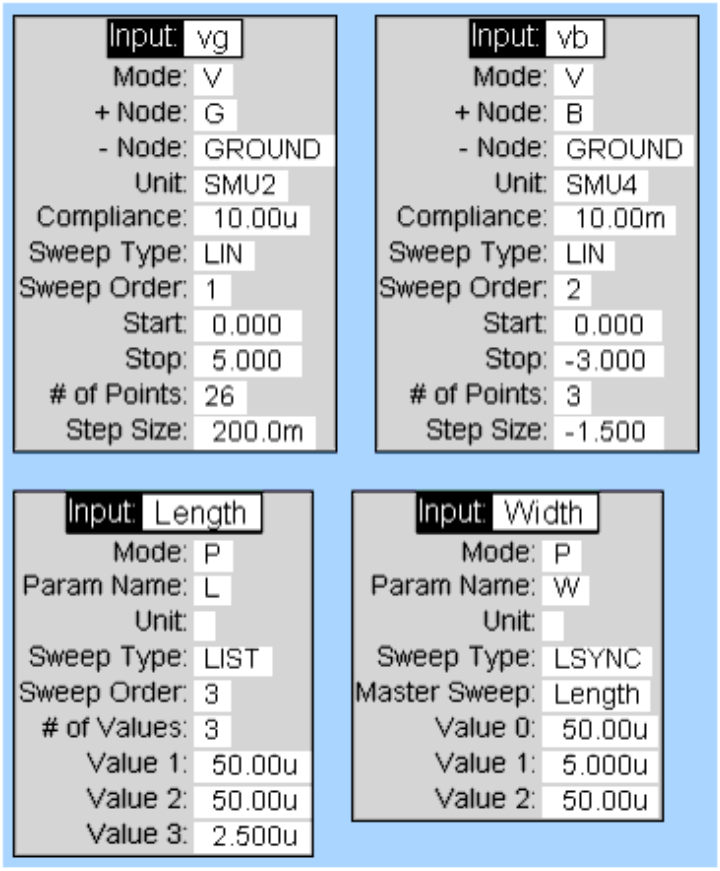
For additional information on sweeping parameters, refer to *Specifying Parameter or Variable Sweeps* (simulation).

The following sections of this topic describe in detail the syntax for the ADS Simulator.

Using LSYNC sweeps

When you use LSYNC sweeps for an ADS simulation, data is written to a data access component file. This file contains the synchronized lists and the specific elements in the netlist that refer to LSYNC values. These elements are accessed using variables and an index.

A single sweep for the LSYNC group is then created to sweep the index.



Example Netlist:

```
Options ASCII_Rawfile=no UseNutmegFormat=yes
model nmos3 MOSFET nmos=yes \
UO          = 1000      \
Vto         = 1.136     \
Nfs         = 0         \
Tox         = 1E-007    \
Nsub        = 5.31E+015 \
Nss         = 0         \
Vmax        = 1E+006    \
Rs          = 0         \
Rd          = 0         \
Rsh         = 0         \
Cbd         = 0         \
Cbs         = 0         \
Cj          = 0         \
Mj          = 0.5       \
Cjsw        = 0         \
Mjsw        = 0.33      \
Is          = 1E-014    \
```

```
Pb          = 0.8       \
Fc          = 0.5       \
Xj          = 9.438E-008 \
Ld          = 2.955E-007 \
Delta       = 0.9338    \
Theta       = 0.04124   \
Eta         = 0         \
Kappa       = 0.2
nmos3:devckt n1 n2 n3 n4 L = lsync0_0 W = lsync0_1 Ad = 1e-10
As = 1e-10 Pd = 0.000104 Ps = 0.000104 ; START SOURCES
V_Source:VGGROUND n2 0 Vdc=0 V_Source:VBGROUND n4 0 Vdc=0
V_Source:VDGROUND n1 0 Vdc=0.1 V_Source:VSGROUND n3 0 Vdc=0
; END SOURCES
INDEX0=0
#uselib "ckt" , "DAC"
DAC:DAC0 File="c:\ictmp\IC19_lsync0" Type="dscr"
InterpMode="index_lookup" iVar1=1 iVal1=INDEX0
lsync0_0=file {DAC0, "lval0"}
lsync0_1 = file {DAC0, "lval1"}
SweepPlan:swp1 Start=0 Stop=5 Step=0.2
SweepPlan:swp2 Start=-3 Stop=0 Step=1.5 Reverse=yes
SweepPlan:swp3 Start=0 Stop=2 Step=1
DC:dc1 SweepPlan=swp1 SweepVar="VGGROUND.Vdc"
ParamSweep:ct1 SimInstanceName="dc1" SweepPlan=swp2
SweepVar="VBGROUND.Vdc"
ParamSweep:ct2 SimInstanceName="ct1" SweepPlan=swp3
SweepVar="INDEX0"
```

And the contents of *IC19_lsync0* will be:

```
BEGIN DSCRDATA
% INDEX0 lval0 lval1
0 5e-05 5e-05
1 5e-05 5e-06
2 2.5e-06 5e-05
```

Interpreting this Section

To make this section more accurate and easier to update, much of the information in it is derived directly from the help facility in the ADS Simulator. The parameter information in the help facility has the following format.

```
Parameters:
   name (units)       attributes  description
Attributes:
   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.
```

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+.	
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"

There are 2 other identifiers not in flag format. One is [ ] next to a parameter name and it means that the parameter is structured as an array. The other is (repeatable) appended to the parameter description and it means that the parameter can appear more than once in the same instance. An example is `OutVar`.

General Syntax

In this section, the following typographical conventions apply:

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 (_).

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 the table Fundamental Units.

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 factors in the table Predefined Scale Factors, then use the numerical equivalent; otherwise, if the first character of the scale factor is one of the prefixes in the table Single-character prefixes, the corresponding scaling is applied.

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
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 the previous table.

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 the following table

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 Unrecognized Scale Factors.

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.

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, see section on Functions 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 1 and 2 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 (see Units and Scale Factors).

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

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
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 degrees)
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 (2 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. Each p_n 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. Each p_n 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, a p_n should never decrease. Each p_n

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*]). See Transient Source Functions.

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

The deembed(*x*) function takes an array, *x*, of 4 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 4 (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.

Caution
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*]) See Transient Source Functions.

`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*]) See Transient Source Functions.

`pwl`(...) piecewise-linear function. See Transient Source Functions.

`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:

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'.

Caution
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 2 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*]) See Transient Source Functions.

The `sprintf()` function is similar to the `C` function which takes a format string for argument *s* and a print argument *x* and returns a formatted string (*x* must be a string, an integer, or a real number). 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:

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

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)
```

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)
```

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)
```

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)
```

Arguments for pwl		
Name	Optional	Default
TIME	NO	
T1	NO	
X1	NO	
T2	YES	NONE
X2	YES	NONE
. . .	. . .	. . .
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 4 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 1 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 1 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.

Caution
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:

AC	CPWCPL4
ACPWDS	CPWCTL
ACPWDTL	CPWDS
AIRIND1	CPWEF
Alter	CPWEGAP
Amplifier	CPWG
AmplifierP2D	CPWOC
AntLoad	CPWSC
BFINL	CPWSUB
BFINLT	CPWTL
BJT	CPWTLFG
BR3CTL	CTL
BR4CTL	C_Model
BRCTL	Chain
BROCTL	Chebyshev
Bessel	Connector
BudLinearization	CostIndex
Butterworth	Crossover
C	DC
CAPP2	DF
CAPQ	DFDevice1
CIND2	DFDevice2
CLIN	DF_DefaultInt
CLINP	DF_Value
COAX	DF_ZERO_OHMS
COAXTL	DICAP
CPW	DILABMLC
CPWCGAP	DOE
CPWCPL2	DRC
DefaultValue	JFET
DeviceIndex	L
Diode	LineCalcTest
EE_BJT2	MACLIN
EE_FET3	MACLIN3
EE_HEMT1	MBEND
EE_MOS1	MBEND2
ETAPER	MBEND3
Elliptic	MBSTUB
FDD	MCFIL
FINLINE	MCLIN
FSUB	MCORN
GCPWTL	MCROS
GMSK_Lowpass	MCROSO
GaAs	MCURVE

Gaussian	MCUREVE2
Goal	MGAP
HB	MICAP1
HP_Diode	MICAP2
HP_FET	MICAP3
HP_FET2	MICAP4
HP_MOSFET	MLANG
Hybrid	MLANG6
IFINL	MLANG8
IFINLT	MLEF
INDQ	MLIN
I_Source	MLOC
InitCond	MLSC
InoiseBD	MLYRSUB
MOS9	MSRTL
MOSFET	MSSLIT
MRIND	MSSPLC
MRINDELA	MSSPLR
MRINDELM	MSSPLS
MRINDNBR	MSSTEP
MRINDSBR	MSSVIA
MRINDWNR	MSTAPER
MRSTUB	MSTEE
MS2CTL	MSTEP
MS3CTL	MSTL
MS4CTL	MSUB
MS5CTL	MSVIA
MSABND	MSWRAP
MSACTL	MTAPER
MSAGAP	MTEE
MSBEND	MTEEO
MSCRNR	MTFC
MSCROSS	MextramBJT
MSCTL	Mixer
MSGAP	MixerIMT
MSIDC	Multipath
MSIDCF	Mutual
MSLANGE	NodeSet
MSLIT	NoiseCorr2Port
MSOBND	Noisey2Port
MSOC	Nsample
MSOP	OldMonteCarlo
MSRBND	OldOpt
OldOptim	PC_Corner

OldYield	PC_CrossJunction
Optim	PC_Crossover
OptimGoal	PC_Gap
Options	PC_Line
OscPort	PC_OpenStub
OutSelector	PC_Pad
PCBEND	PC_Slanted
PCCORN	PC_Taper
PCCROS	PC_Tee
PCCURVE	PC_Via
PCILC	PIN
PCLIN1	PIN2
PCLIN10	PLCQ
PCLIN2	ParamSweep
PCLIN3	PinDiode
PCLIN4	PoleZero
PCLIN5	Polynomial
PCLIN6	Port
PCLIN7	PowerBounce
PCLIN8	PowerGroundPlane
PCLIN9	R
PCSTEP	RCLIN
PCSUB	RIBBON
PCTAPER	RIBBON_MDS
PCTEE	RIND
PCTRACE	RWG
PC_Bend	RWGINDF
PC_Clear	RWGT
RWGTL	SLSTEP
R_Model	SLTEE
RaisedCos	SLTL
SAGELIN	SLUCTL
SAGEPAC	SLUTL
SBCLIN	SMITER
SBEND	SOCLIN
SBEND2	SPIND
SCLIN	SS3CTL
SCROS	SS4CTL
SCURVE	SS5CTL
SDD	SSACTL
SL3CTL	SSCLIN
SL4CTL	SSCTL
SL5CTL	SSLANGE
SLABND	SSLIN

SLCQ	SSSPLC
SLCRNR	SSSPLR
SLCTL	SSSPLS
SLEF	SSSUB
SLGAP	SSTEP
SLIN	SSTFR
SLINO	SSTL
SLOBND	SSUB
SLOC	SSUBO
SLOC_MDS	STEE
SLOTTL	S_Param
SLRBND	S_Port
SLSC	ScheduleCycle
Short	VBIC
Substrate	VIA
SweepPlan	VIA2
SwitchV	V_Source
SwitchV_Model	VnoiseBD
TAPIND1	WIRE
TFC	WIRE_MDS
TFC_MDS	Y_Port
TFR	Yield
TFR_MDS	YieldOptim
TL	YieldSpec
TLIN	YieldSpecOld
TLIN4	Z_Port
TLINP	__fdd
TLINP4	__fdd_v
TL_New	_ac_state
TQAVIA	_c1
TQCAP	_c10
TQFET	_c11
TQFET2	_c12
TQIND	_c13
TQRES	_c14
TQSVIA	_c15
TQSWH	_c16
TQTL	_c17
Tran	_c18
UFINL	_c19
UFINLT	_c2
Unalter	_c20
_c21	_freq6

_c22	_freq7
_c23	_freq8
_c24	_freq9
_c25	_gaussian
_c26	_gaussian_tol
_c27	_get_fnom_freq
_c28	_get_fund_freq_for_fdd
_c29	_harm
_c3	_hb_state
_c30	_i1
_c4	_i10
_c5	_i11
_c6	_i12
_c7	_i13
_c8	_i14
_c9	_i15
_dc_state	_i16
_default	_i17
_discrete_density	_i18
_divn	_i19
_freq1	_i2
_freq10	_i20
_freq11	_i21
_freq12	_i22
_freq2	_i23
_freq3	_i24
_freq4	_i25
_freq5	_i26
_i27	_sopt
_i28	_sp_state
_i29	_sv
_i3	_sv_bb
_i30	_sv_d
_i4	_sv_e
_i5	_tn
_i6	_to
_i7	_tr_state
_i8	_tt
_i9	_uniform
_lfsr	_uniform_tol
_mvgaussian	_v1
_mvgaussian_cov	_v10
_n_state	_v11
_nfmin	_v12

_p2dInputPower	_v13
_phase_freq	_v14
_pwl_density	_v15
_pwl_distribution	_v16
_randvar	_v17
_rn	_v18
_shift_reg	_v19
_si	_v2
_si_bb	_v20
_si_d	_v21
_si_e	_v22
_sigproc_state	_v23
_sm_state	_v24
_v25	conj
_v26	cos
_v27	cos_pulse
_v28	cosh
_v29	cot
_v3	coth
_v30	coupling
_v4	ctof
_v5	ctok
_v6	cxform
_v7	d_atan2
_v8	damped_sin
_v9	db
_xcross	dbm
abs	dbmtoa
access_all_data	dbmtov
access_data	dbmtow
aele	dbpolar
and	dbwtow
arcsinh	dcSourceLevel
arctan	deembed
atan2	define
awg_dia	deg
bin	delay
bitseq	dep_data
boltzmann	deriv
by	discrete
c0	distcompname
complex	doe
doeindex	generate_qam16_spectra
dphase	generate_qpsk_pulse_spectra

dsexpr	get_array_size
dstoarray	get_attribute
e	get_block
e0	get_fund_freq
echo	get_max_points
else	global
elseif	globalnode
end	ground
endif	hugereal
equals	i
erf_pulse	if
eval_poly	ilsb
exp	imag
exp_pulse	index
file	innerprod
fread	inoise
freq	int
freq_mult_coef	internal_generate_gmsk_iq_spectra
freq_mult_poly	internal_generate_gmsk_pulse_spectra
ftoc	internal_generate_piqpsk_spectra
ftok	internal_generate_pulse_train_spectra
gauss	internal_generate_qam16_spectra
gaussian	internal_generate_qpsk_pulse_spectra
generate_gmsk_iq_spectra	internal_get_fund_freq
generate_gmsk_pulse_spectra	internal_window
generate_piqpsk_spectra	interp
generate_pulse_train_spectra	interp1
interp2	names
interp3	nested
interp4	nf
iss	nfmin
itob	no
iusb	nodoe
jn	noisefreq
ktoc	noopt
ktof	norm
lbtran	nostat
length	not
limit_warn	notequals
list	omega
ln	opt
ln10	optIter
local	or
log	parameters

logNodesetScale	phase
logRshunt	phase_noise_pwl
log_amp	phasedeg
log_amp_cas	phaserad
mag	planck
makearray	polar
max	polarcpx
mcTrial	ppt
mcindex	pulse
min	pwl
model	pwlr
multi_freq	qelectron
qinterp	sprintf
rad	sqrt
ramp	ssfreq
randtime	stat
rawtoarray	step
read_data	strcat
read_lib	stypexform
readdata	sym_set
readlib	system
readraw	tan
real	tanh
rect	temp
rem	tempkelvin
ripple	thd
rms	then
rn	time
rpsmooth	timestep
scalearray	tinyreal
sens	to
setDT	toi
sffm	tranorder
sgn	transform
sin	u0
sinc	unconst
sine	unicap
sinh	uniform
sink	v
sopt	value
sourceLevel	vlsb
vnoise	
vss	

vswrpolar	
vusb	
window	
yes	

SPECTRE Simulator

This section describes the details of using the SPECTRE simulator with IC-CAP. For general information on IC-CAP simulation, refer to *Simulation* (simulation).

SPECTRE Interfaces

SPECTRE is a SPICE-like circuit simulator developed by Cadence Design Systems that simulates analog and digital circuits at the differential equation level using direct methods.

SPECTRE uses the same basic algorithms used in UCB SPICE but the implementation of these algorithms uses the most up-to-date methods currently available.

IC-CAP offers 3 different interfaces for use with the SPECTRE simulator:

- SPECTRE
- SPECTRE443
- SPECTRE442

SPECTRE Interface

The SPECTRE interface is compatible with SPECTRE version 4.4.3 simulators and later. Unlike the SPECTRE443 and SPECTRE442 interfaces which invoke the SPICE netlist parser, this interface uses native SPECTRE netlist syntax to parse data from the circuit page. This alleviates the need to translate SPECTRE netlists to SPICE syntax prior to entering the netlists on the circuit page. See the following section, Valid SPECTRE Netlist Syntax for IC-CAP.

SPECTRE443 Interface

This interface is compatible with SPECTRE versions up to 5.0.0. The SPECTRE443 interface invokes a SPICE netlist parser, unlike the SPECTRE implementation which uses native SPECTRE netlist syntax to parse data from the circuit page. This interface requires that SPECTRE netlists are first converted to SPICE syntax prior to entering them on the circuit page.

SPECTRE442 Interface

This interface is compatible with SPECTRE simulator version 4.2.2 only. The SPECTRE442 interface invokes the SPICE netlist parser, unlike the SPECTRE interface which uses native SPECTRE netlist syntax to parse data from the circuit page. This interface requires that SPECTRE netlists are first converted to SPICE syntax prior to entering them on the circuit page.

Caution
The SPECTRE442 interface is no longer recommended. IC-CAP is only tested against the latest version of SPECTRE. The SPECTRE442 interface is documented only to assist in migrating to the SPECTRE443 or SPECTRE interface.

Open Simulator Interface (OSI)

This interface requires the compilation of a translation module (see spectre3.c in $ICCAP_ROOT/src). This translation module allows IC-CAP to operate as though it is interfacing to SPICE 3. This interface is no longer recommended, but is documented to help migration efforts from the old interface to the new SPECTRE interface template. For details, see Using Template SPICE3 and the Open Simulator Interface spectre3.c.

Circuit Model Descriptions

The following section describes the type of circuit page netlists required when using the SPECTRE interface. Please refer to Circuit Model Descriptions for the netlist requirements for the SPECTRE443, SPECTRE442, or the SPICE3 OSI interfaces.

For valid circuit syntax descriptions, see the Cadence SPECTRE simulator user's documentation.

Specifying Simulator Options

For information on available simulator options and their syntax, refer to the Cadence SPECTRE simulator user's documentation.

Simulator options are specified in the first line of the circuit definition using the following syntax:

```
options OPT1 = OPTVAL1 OPT2 = OPTVAL2 ... OPTN = OPTVALN
```

where

OPT denotes the option keyword used by the simulator.

OPTVAL is the corresponding option value. Some options do not require a value. This field may or may not be specified, depending on the option.

A space is the only delimiter required between options.

The nominal and operating temperatures, TNOM and TEMP, are commonly used options. TNOM is the temperature at which the model parameters are extracted. TEMP is the temperature at which the simulation is performed.

Note
When performing an optimization to extract model parameters, TEMP and TNOM should be set to the same value so that simulations during optimization are performed at TNOM. TNOM must be defined to guarantee consistency between simulation and extraction.

You can also specify these variables by entering a value (in °C) for the global variables TNOM and TEMP in the System Variables table in the Utilities application.

In general, TNOM and TEMP can be in any variable table, allowing different Models, DUTs or Setups to use different nominal and operating temperatures.

IC-CAP checks for these global variables before running a simulation. If it does not find

the variable, IC-CAP uses the value set in the Circuit Editor options statement. Otherwise, IC-CAP analyzes the circuit using the simulator's default values.

Valid SPECTRE Netlist Syntax for IC-CAP

The SPECTRE interface parses netlists written in native SPECTRE syntax.

During a simulation using the SPECTRE template, IC-CAP examines the netlist entered on the Circuit page for:

- The name of the device to be modelled
- The external nodes of the device
- The model-level parameters
- The device-level parameters

IC-CAP is intended for single-device model extractions. Therefore, not all valid SPECTRE netlists are accepted by IC-CAP.

Valid SPECTRE Constructs

IC-CAP uses 3 SPECTRE constructs:

- the device statement
- the subcircuit (subckt) block
- the model statement

Valid SPECTRE Circuit Page Configurations

There are 3 valid Circuit page configurations:

- A single device statement *and* a single model card
- A single subcircuit block
- A single device statement followed by a single subcircuit block

> **Note**
> Other supporting statements can be added in and around the configurations mentioned above. This includes all valid SPECTRE syntax statements *other than* the device, subckt, and model statements. These 3 constructs are limited in number and combination as described above.

Describing a Device

A device statement describes a single SPECTRE element of any type. The general form of device statement is:

```
DNAME NODE1 NODE2...NODEN MNAME DPAR1=DVAL1 DPAR2=DVAL2
```

where

DNAME is the device name with the first letter being a simulator-defined key letter, denoting the type of model being specified.

NODE denotes the node name for the device connection.

MNAME is the name of a built-in device, or the name of a model or subcircuit definition. This is the same *MNAME* specified in the model definition described below.

DPAR is a predefined DUT parameter name.

DVAL is the specified DUT parameter value. Refer to the SPICE Reference for the DUT parameter names available for each model.

A plus sign (+) that appears as the first character of a line or a back slash (\ ) that appears as the last character in a previous line denotes a continuation of the previous line. This continuation character is often used for easier readability when specifying the *model* card.

Describing the Model

A model definition specifies the parameters of a particular model that is referenced by a device statement (see Describing a Device). When a parameter is not specified, the default value in the model is used. The general form of the model definition is:

```
model MNAME TYPE PNAME1=PVAL1 PNAME2=PVAL2 ...PNAMEN=PVALN
```

where

MNAME is the model name. Regardless of the model name entered in the MNAME field of the model definition statement, IC-CAP substitutes the name of the Model as it is called in the Model List when the simulator input deck is built.

Note
Noise is a reserved word in SPECTRE and must not be used in naming components of the netlist. Do not use the name *"noise"* for DUTs or Models. IC-CAP substitutes the Model/DUT name for the name in the Circuit or Test Circuit folders respectively.

TYPE is a valid SPECTRE component type.

PNAME is a parameter name for the particular model type.

PVAL is the parameter value.

A plus sign (+) that appears as the first character of a line or a back slash (\ ) that appears as the last character in a previous line denotes a continuation of the previous line. This continuation character is often used for easier readability when specifying the *model* card.

Describing Subcircuits

A subcircuit model is used to describe a circuit that contains more than 1 element.

The syntax is similar to the syntax in SPICE. The subcircuit description must begin with a

subckt and end with an *ends* declaration. Statements between these 2 declarations describe the subcircuit components.

The general form of a subcircuit definition is:

```
subckt SUBNAME (NODE1 NODE2...NODEN)
       parameters PAR1=PARVAL1 PAR2=PARVAL2 ...PARN=PARVALN
       <subcircuit devices and/or models listed here>
       ends SUBNAME
```

where

SUBNAME is the subcircuit name. Regardless of the subcircuit name entered in the SUBNAME field of the *subckt* definition statement, IC-CAP substitutes the name of the Model being simulated when the simulator input deck is built.

Note
Noise is a reserved word in SPECTRE and must not be used in naming components of the netlist. Do not use the name *"noise"* for DUTs or Models. IC-CAP substitutes the Model/DUT name for the name in the Circuit or Test Circuit folders respectively.

NODE denotes the node name for the device connection.

PAR1 ... PARN are subcircuit parameters that can be passed through subcircuit calls. If a subcircuit is used in conjunction with a device statement, then the parameters specified on the device line will also need to be listed here. In this case, those parameters are added to the DUT Parameters table. All other parameters not listed in the device statement will be added to the Model Parameters table. If the subcircuit description is used without an associated device statement, then all parameters listed here will be entered in the DUT Parameters table.

PARVAL1 ... PARVALN are the corresponding parameter values. Depending on the context (see previous paragraph), these parameters become either DUT parameters or model parameters which can be modified in the DUT Parameters table of the Model Parameters table.

The body of the subcircuit model description contains the components of the subcircuit using element and *model* statements.

Using a Device Statement and Model Card Configuration

The device statement and model card is the simplest circuit page configuration. The template parses the model card into the Model Parameters page and the device parameters into the DUT Parameters page. The device statement provides the external nodes.

Example syntax:

```
q1 C B E S NPN area = 1.0
model NPN bjt
+ is = 1E-16
```

```
+ bf = 100
```

In this case, *is* and *bf* will appear on the Model Parameters page, and *area* will appear in the DUT Parameters page.

Note
The device statement and model card may appear in any order.

Using a Single Subcircuit Block Configuration

This circuit page configuration interprets the subcircuit as a single device. If the subcircuit includes a Parameters statement, the template parses these parameters as device parameters, where they appear in the DUT Parameter Table. All parameters on model or device statements within the subcircuit appear in the Model Parameter Table in the form:

```
<inst/model>.<parameter>
```

Example syntax:

```
subckt realnpn (C B E)
parameters area=1
LE E 4 inductor l=.35n
LB B 5 inductor l=.2n
CC C 0 capacitor c=.255p
Q1 C 5 4 NPN area = area
model NPN bjt
+ is = 1E-16
+ bf = 100
ends realnpn
```

In this case, *LE.l*, *LB.l*, *CC.c*, *NPN.is*, and *NPN.bf* will appear in the Model Parameters table and *area* will appear in each DUT Parameters table.

Note
Note, *Q1.area* does not appear because its value is not a simple number. IC-CAP only identifies parameters with simple numbers for extraction.

When this circuit is simulated, IC-CAP outputs the subcircuit as well as an device statement to call the subcircuit.

See the example file *model_files/bjt/spectre_ncehf.mdl* for a working model.

Using a Device Statement Followed by a Subcircuit Block

In some situations, you must extract parameters from a device defined by a subcircuit whose parameters listed in the Parameters statement within the subcircuit are your *model* parameters and not your *device* parameters. Use the "device statement followed by a subcircuit block" configuration.

In this configuration, all parameters listed with the subcircuit *parameters* statement are parsed as *model* parameters, unless they are referenced on the device statement, in which case they are treated as *device* parameters.

Example syntax:

```
q1 C B E S realnpn area=1.0
subckt realnpn C B E S
parameters area=1.0 is=1e-16 bf=100 lb=1
lb1 B 1 inductor l=lb
q1 C 1 E S NPN area=area
model NPN bjt is=is bf=bf
ends realnpn
```

In the this example, there are 3 model parameters, *is*, *bf* and *lb*, and 1 device parameter, area.

Note the difference between this configuration and the single-subcircuit configuration which has only a *subckt* definition and no device.

Test Circuits and Hierarchical Simulation

When characterizing a circuit, it is often necessary to add circuitry around a circuit or device to model the actual measurement Setup. IC-CAP provides a Test Circuit Editor to allow modeling of this additional bias circuitry. Select the DUT from the DUT/Setup panel. Click the Test Circuit tab and enter the test circuit description in the same manner you would enter a Circuit Description. The test circuit definition should include a call to the device or subcircuit defined in the Circuit Editor, as well as the additional circuitry needed to model the external parasitics of the measurement Setup.

Note
When you define a test circuit, the DUT Parameter table contains the values specified in the test circuit specification. Regardless of the subcircuit name entered in the SUBNAME field of the *subckt* declaration, IC-CAP uses the name of the DUT being simulated when the simulator input deck is built.

Note
Noise is a reserved word in SPECTRE and must not be used in naming components of the netlist. Do not use the name *"noise"* for DUTs or Models. IC-CAP substitutes the Model/DUT name for the name in the Circuit or Test Circuit folders respectively.

Subcircuit and device model specifications can be called from inside another model. This enables you to perform hierarchical simulations to study a circuit at different levels.

When making reference to another model, the model name must be used as it appears in the IC-CAP Model List. For example, assume you have defined 3 models, *model1*, *model2*, and *model3*. *model1* has a circuit model description that is a device definition. The circuit model description for *model2* is a subcircuit definition at the gate level that includes a call to *model1* in a device call statement. And, the circuit model description for *model3* is a subcircuit definition that includes a call to *model2* in a subcircuit call statement. When you simulate a Setup in *model3*, IC-CAP traverses the Model hierarchy and uses the circuit model description defined in *model3*, which includes calls to *model1* and *model2*. The syntax for calling a device model is identical to that described in the *Device Model Description* section.

The general form of the device call is:

```
DNAME NODE1 NODE2...NODEN MNAME DPAR1=DVAL1 DPAR2=DVAL2
```

Calling a subcircuit specification allows you to insert an entire subcircuit into a circuit as if it were a single component. The call requires a syntax identical to that used in SPECTRE. The general form of the subcircuit call is:

```
DNAME NODE1 NODE2...NODEN SUBNAME DPAR1=DVAL1 DPAR2=DVAL2
```

where

DNAME is the name of the subcircuit call statement. The only requirement for this name is that it must start with the letter D.

NODE denotes the node name for the device connection.

SUBNAME is the name of the subcircuit, previously described by a *subckt* definition. This must have the name of the model as it appears in the Model List if it is in a different model.

DPAR are passed in the parameter names.

DVAL are subcircuit parameter values. The order in which they are listed in the subcircuit call statement must match the parameters list in the subcircuit definition.

Note
When a test circuit is included in the Model, IC-CAP uses the test circuit description as the top level circuit definition. The node number connections defined in the test circuit description, not the circuit description, are used as the external nodes. Because of this, any node-number-to-node-name cross-referencing in the circuit description is not used. Only node names equated to node numbers in the test circuit description can be used when specifying Inputs and Outputs in the Setup Editor. When only node numbers are specified in the test circuit description, (that is, they are not equated to node names) these same node numbers must be used in the Input and Output node fields.

Piped and Non-Piped SPECTRE Simulations

The following sections describe the differences in piped and non-piped simulations for the various SPECTRE simulators. Each section also describes the argument syntax required to invoke each of the template simulators. This information is needed when writing the user translation module, since these are the arguments supplied by IC-CAP when it calls the translation module. For information on the translation module and adding a simulator, refer to *Adding a Simulator* (simulation).

There are 3 methods you can use to link to the SPECTRE simulator interface:

- Use template SPECTRE, SPECTRE443, or SPECTRE442 with CANNOT_PIPE.
- Use template SPECTRE, SPECTRE443, or SPECTRE442 with CAN_PIPE.
- Use template SPICE3 and the Open Simulator Interface _spectre3.c.

Note
The methods using SPECTRE or SPECTRE442/443 offer significant speed enhancements with some minor features that will not work properly. Be sure to read the following sections describing their limitations.

Note
The method using SPICE3 is fully supported, but offers the slowest speed. It is not recommended, except when methods using SPECTRE or SPECTRE442/443 do not work, or are unavailable.

Using SPECTRE Simulator Templates with CANNOT_PIPE

If you specify the template SPECTRE, SPECTRE442 or SPECTRE443, you can greatly speed up your simulations. This template will use the SPECTRE *alter* command to simulate multiple bias steps in 1 simulation. This improves many multi-sweep simulations such as an S-parameter setup with 2 sweeps. Using the Open Simulator Interface method, each of these bias steps would require a separate simulation.

The one known limitation with this method is that parameter sweeps will not work properly with certain parameters that are declared in a subcircuit at the Circuit page level when a Test circuit is being used. Parameters that are declared with an "=" sign will work even under this configuration, but parameters that are declared without an "=" sign will not work. In the following example, parameter sweeps will work for IS, but not for R1.

Circuit Page:

```
subckt CIRC  1=A  2=C
R1 1 2 50
Q1 1 2 1 2 NPN
model NPN BJT IS=10e-15
.ENDS
```

Test Circuit:

```
subckt CIRC2  1=A  2=C
XTEST 1 2 CIRC
.ENDS
```

Using SPECTRE Simulator Templates with CAN_PIPE

IC-CAP may not work properly with parameters defined using *$mpar()* in *#echo* lines. If using such a circuit, Agilent Technologies does not recommend using CAN_PIPE. Use CANNOT_PIPE instead.

Specifying CAN_PIPE with SPECTRE, SPECTRE442 or SPECTRE443 templates will use a mode that will allow the simulator to stay up for multiple simulations of the same setup as long as the only thing changing are parameters. This is what happens during an optimization which has all targets within 1 setup. This mode is not officially supported by Cadence, so use the link at your own risk. Our testing has shown it to provide significant performance improvements.

Limitations of this method include:

- This mode has the same limitation described in the previous section.
- If a Test circuit is used, this mode offers no performance enhancement.
- This mode does not work with remote hosts.

Using Template SPICE3 and the Open Simulator Interface spectre3.c

> **Note**
> Using Template SPICE3 requires more processing time than the other SPECTRE templates. Using Template SPICE3 is not recommended, except when methods using SPECTRE, SPECTRE442, or SPECTRE443 are unavailable.

Using IC-CAP's Open Simulator Interface, a C-language *Translation Module* is provided that makes SPECTRE simulation capability available in IC-CAP. This module and instructions for performing SPECTRE simulations in IC-CAP are described here. For general information on the Open Simulator Interface, refer to *Adding a Simulator* (simulation).

The IC-CAP/SPECTRE link uses UCB SPICE3 as the *template* simulator. When performing a SPECTRE simulation in IC-CAP, IC-CAP behaves as if it is performing a SPICE3 simulation. Therefore it generates an input deck in SPICE3 format, calls the simulator and reads back a binary raw data file in SPICE3 format. Through the Open Simulator Interface, the call to the simulator is actually calling the executable version of the C-language *Translation Module, spectre3.c.* This executable, called *spectre3,* translates the SPICE3 input deck to a SPECTRE input format, calls SPECTRE to perform the simulation, then translates the SPECTRE format binary raw data file to SPICE3 format which is read by IC-CAP. The source code file *spectre3.c* is located in the *$ICCAP_ROOT/src* directory.

> **Note**
> When using SPECTRE, the CDS_LICENSE_DIR environment variable must be set. This variable contains the directory path for the license file required by the SPECTRE simulator. Refer to the SPECTRE Reference Manual for detailed procedures on installing the SPECTRE simulator.

> **Note**
> SPECTRE does not support a secondary sweep in the DC specification. For DC simulations, set the System Variable MAX_DC_SWEEPS to 1 so that IC-CAP generates a separate input deck for every point in the secondary sweep, if it exists.

> **Note**
> If you set the SPECTRE variable SPECTRE_DEFAULTS in your system startup file, for example, the . *profile* file, do not use the -E option. Use the following sntax:
>
> ```
> SPECTRE_DEFAULTS +l %C.r.out -f psfascii
> ```

To set up SPECTRE simulation capability in IC-CAP:

1. Add the *spectre_simulator to the _usersimulators* file in the directory *$ICCAP_ROOT/iccap/lib*, as shown next.
   ```
   spectre spice3 /<your path>/spectre3 "<host_machine_name>" CANNOT_PIPE
   ```
2. The *host_machine_name* is the host computer for the SPECTRE simulations. This name can be left blank ("") if SPECTRE and IC-CAP are running on the same computer. Since SPECTRE does not have the ability to perform piped simulations in IC-CAP, the CANNOT_PIPE flag must always be set, as shown in the above example.

3. Make the following change to the *spectre3.c* program to customize it for your environment:
   - In the *main* routine, specify the full pathname of the actual SPECTRE simulator on your system.
4. Compile the translation module using the following command:
   `cc -o spectre3 spectre3.c -lm`
5. Move the executable file, *spectre3* to a permanent location such as *$ICCAP_ROOT/bin.* The location must match the path specified in the *usersimulators* file.
6. In IC-CAP, set the SIMULATOR variable to *spectre* or specify *spectre* with the *Select Simulator* command in the IC-CAP Tools Menu.

The following files are generated in your home directory when running SPECTRE simulations in IC-CAP:

- *spectre.cki* - SPECTRE format circuit description deck file translated from the SPICE3 circuit description deck.
- *spectre.raw* - SPECTRE formatted binary raw data output file generated by a SPECTRE simulation.
- *spectre.log* - Output print file generated by a SPECTRE simulation.

spectre.raw and *spectre.log* are automatically removed from your home directory after the simulation is completed in IC-CAP.

Note
Some of the new models implemented in SPECTRE use slightly different syntax for the model statement than they would for SPICE3. This difference will not be accounted for by the translator; you must change the model statement in the Circuit Description folder before simulating. The following examples show how the model statement would read for the MM9 and BSIM3 models:

```
model <name> mos902 type=n <parameters>
model <name> bsim3 type=n <parameters>
```

HSPICE Simulator

HPSPICE The Agilent Technologies implementation of SPICE2. Although there are some differences between this version and the SPICE2G.6 version from U.C. Berkeley, these 2 simulators are compatible. For more information refer to *SPICE Simulator Differences* (simulation). The version of HPSPICE provided with IC-CAP can be run only from within the IC-CAP program-it cannot be run stand-alone.

Note
The HSPICE simulator, developed by Synopsys, uses input deck syntax similar to that of the SPICE-type simulators; thus, it is referred to as a SPICE-type simulator in this manual. IC-CAP currently supports only the features of HSPICE also available in the U.C. Berkeley SPICE simulators.

Piped, Non-Piped, and Client/Server HSPICE Simulations

Note
The CAN_PIPE token is supported for HSPICE in user simulators. This token can now be used on local Linux HSPICE and local Solaris HSPICE simulations with HSPICE-2007.03.SP1. It is not a true piped mode (netlists and raw files are still written to disk), but provides substantial performance improvement by using an interactive mode that avoids restarting HSPICE for every simulation. Beginning in HSPICE 2008.03-SP1, HSPICE license will time out in 1800 seconds for CAN_PIPE mode. You can customize the license timeout by setting variable HSPICE_LICENSE_TIMEOUT (unit by second).

Non-piped HSPICE simulations are identical to non-piped SPICE simulations. This type of simulation is performed when the Simulation Debugger is set to ON. If CANNOT_PIPE is specified for HSPICE, even when the Simulation Debugger is OFF, it still performs a non-piped simulation. This means that HSPICE must be restarted for every simulation. Because of this, there is no noticeable difference in simulation speed when the Simulation Debugger is set to ON or OFF.

Syntax: Piped HSPICE Simulations

The command format for an HSPICE piped simulation is as follows:

```
hspice -I
```

load deckname and *run* commands are then passed to the running HSPICE process.

Syntax: Non-Piped HSPICE Simulations

The command format for an HSPICE non-piped simulation is as follows:

```
hspice -i deckfile -o logfile
```

where

deckfile is the input deck file containing the circuit description and analysis commands.

logfile is the listing of information about the simulation generated by HSPICE. If the simulation debugger is open, this file will be displayed in the *Output Text* portion of the simulation debugger.

The output binary data file is written to a file named *deckfile.suffix* where *suffix* depends on the type of analysis being performed. Refer to the *HSPICE User's Manual* for more information.

Syntax: Client/Server mode HSPICE Simulations

On Windows, CAN_PIPE is not supported, but HSPICE provides a method of invoking a standing server process to access HSPICE licenses. If this was launched via the *hspui* program, IC-CAP can simulate faster by launching HSPICE with the following syntax:

```
hspice -C deckfile -o logfile
```

where

deckfile is the input deck file containing the circuit description and analysis commands.

logfile is the listing of information about the simulation generated by HSPICE. If the simulation debugger is open, this file will be displayed in the *Output Text* portion of the simulation debugger.

However, if the server has not been started, the simulation still occurs but at a slower speed.

To configure IC-CAP to send the *-C* instead of *-i*, specify the template name *hspice-C* as the second field in your usersimulators line example:

```
hspiceC hspice-C c:\synopsys\Z-2007.09\bin\hspice.exe ""
CANNOT_PIPE
```

Circuit Model Descriptions

The circuit description for the HSPICE simulator is similar to the UCB SPICE simulator circuit description. Refer *Circuit Model Descriptions* (simulation) described under section SPICE Simulators.

Circuit Description Syntax

Basic HSPICE syntax rules are the same as SPICE-type simulators. Refer to *Circuit Description Syntax* (simulation) basic syntax rules for creating a circuit description. Refer to HSPICE User's Manual for complete syntax rules.

Note
Before performing HSPICE simulations, specify the HSPICE version name in the System Variable HSPICE_VERSION. If this variable is not specified, IC-CAP will assume the latest version of HSPICE is being used.

SPICE Simulators

This section describes the details of using the SPICE simulators with IC-CAP. For general information on IC-CAP simulation, refer to *Simulation* (simulation).

Note
The Solaris OS must include the cpp utility, which IC-CAP uses to manage output from SPICE simulators. See "System Requirements" in the *Installation and Configuration Guide* for more details.

IC-CAP can interface with the following SPICE simulators. They are provided as a courtesy to IC-CAP users (though not supported by Agilent Technologies).

The SPICE simulators support the following analysis types:

- DC
- AC
- Transient
- Noise
- Capacitance Voltage (CV)
- 2-Port (S,H,Y,Z,K,A parameter)
- Time-Domain Reflectometry (TDR)

  Note
  The latter 3 simulation types are not directly available in the SPICE simulators; IC-CAP builds the additional circuitry required in the simulator input files to perform the simulation.

IC-CAP supports the features of ELDO that are also available in the UCB SPICE simulators but also provides limited support for models written in either ELDO-FAS or HDL-A. ELDO is an analog simulator developed by Mentor Graphics Corp. ELDO input deck syntax is compatible with that of the SPICE type simulators; therefore, in ELDO is categorized as a SPICE-type simulator this manual.

The IC-CAP version of SPICE3 supports the following models:

Model Group	Supported Models	Model Files
MOSFET	Level 1, Level 2, Level 3	nmos/pmos2 nmos3/pmos3
	BSIM3, BSIM4	BSIM3_DC_CV_Measure BSIM3_DC_CV_Extract BSIM3_RF_Measure BSIM3_RF_Extract BSIM3_AC_Noise_Tutorial BSIM3_CV_Tutorial BSIM3_DC_Tutorial BSIM3_Temp_Tutorial BSIM3_DC_CV_Finetune BSIM4_DC_CV_Measure BSIM4_DC_CV_Extract BSIM4_RF_Measure BSIM4_RF_Extract BSIM4_DC_CV_Tutorial BSIM4_DC_CV_Finetune
	MOS Model 9	mm9 mm9_demo
BJT	Gummel Poon	bjt_npn/bjt_pnp bjt_nhf bjt_ncehf bjt_ft mnsnpn sabernpn
GaAs	Statz	UCBGaas UGaashf
Diode	PN Diode	pn_diode
	Philips JUNCAP	juncap

The following additional SPICE-like simulators are also discussed in this section:

- PRECISE
- PSPICE

SPICE Simulation Example

The circuit description is predefined for all IC-CAP configuration files. Enter this description if a new model is being defined; edit the description to fit specific needs. The syntax is identical to the syntax used for describing circuits in a typical SPICE simulation deck.

This simulation example will use the IC-CAP supplied Model *bjt_npn.mdl*.

1. Select the simulator by choosing **Tools > Options > Select Simulator > spice2**. Choose **OK**.
2. Choose **File > Open > bjt_npn.mdl**. Choose **OK**.
3. View the circuit description by clicking the **Circuit** tab.
   The circuit description is shown below in Circuit Description Deck for an NPN Bipolar Transistor. This deck describes the circuit (in this case, a single device) to be used in the simulation.
4. To view input and output for the fearly setup, click the **DUTs-Setups** tab and select **fearly**.

The Measure/Simulate folder appears with the inputs vb, vc, ve, and vs, and the output ic. The vc input specifies a voltage source at node C that sweeps linearly from 0 to 5V in 21 steps. The ic output specifies that current at node C be monitored.
In the Plots folder, icvsvc is specified so that the results of the simulation can be viewed graphically.

5. To simulate, click the **Simulate** button in the Measure/Simulate folder. The Status line displays Simulate in progress.
When the simulation is complete, the Status line displays IC-CAP Ready.
6. To view the results of the simulation, display the Plots folder and click **Display Plot**. The plot displays measured data represented by solid lines and simulated data represented by dashed lines.

> **Note**
> For syntax examples of running a remote simulation, refer to *Remote Simulation Examples* (simulation).

Circuit Description Deck for an NPN Bipolar Transistor

```
Q1   1 = C 2 = B 3 = E 4 = S NPN AREA = 1.0
.MODEL NPN NPN
+ IS = 36.76e-18
+ BF = 336.1
+ NF = 1.003
+ VAF = 35.25
+ IKF = 22.07m
+ ISE = 1.882f
+ NE = 1.932
+ BR = 4.103
+ NR = 1.005
+ VAR = 1.651
+ IKR = 147.3u
+ ISC = 15.69f
+ NC = 1.857
+ RB = 522.0
+ IRB = 61.43u
+ RBM = 1.000m
+ RE = 8.435
+ RC = 57.05
+ XTB = 1.700
+ EG = 1.110
+ XTI = 3.000
+ CJE = 44.06f
+ VJE = 871.7m
+ MJE = 429.9m
+ TF = 10.49p
+ TR = 1.700m
+ XTF = 247.4
+ VTF = 1.622
+ ITF = 140.6m
+ PTF = 218.8
+ CJC = 68.94f
+ VJC = 603.8m
+ MJC = 290.6m
+ XCJC = 300.0m
+ TR = 1.700n
+ CJS = 111.9f
+ VJS = 465.0m
```

```
+ MJS = 241.9m
+ FC = 500.0m
```

Piped and Non-Piped Simulations

The following sections describe the differences in piped and non-piped simulations for the various SPICE simulators. Each section also describes the argument syntax required to invoke each of the template simulators. This information is needed when writing the user translation module, since these are the arguments supplied by IC-CAP when it calls the translation module. For information on the translation module and adding a simulator, refer to the section *Adding a Simulator* (simulation).

Piped and Non-Piped SPICE Simulations

A non-piped simulation receives the input deck information from a file, performs the simulation and sends the binary output data and resulting text output to other files. The simulator process is restarted for every simulation.

A piped simulation receives the input deck information from a pipe connected to *standard input*, performs the simulation and sends the output data to a pipe connected to *standard output*. The simulator process will remain on until another simulator is selected. Setting the RETAIN_SIMU variable to TRUE overrides this behavior and allows multiple simulators to remain running. This uses additional memory but increases speed when frequently switching between simulators. In all cases, a piped simulator process will be turned off when the Simulation Debugger is turned on.

The text output from a simulation usually contains an explanation of any errors which may have been encountered during the simulation. Piped simulations do not save any text output from the simulation. If an error occurs during a piped simulation, IC-CAP issues a message in an error box stating that an error has occurred and recommending that the simulation be repeated with the Simulation Debugger turned on. IC-CAP performs non-piped simulations when the Simulation Debugger is ON.

In general, piped simulations are faster than non-piped simulations for any given simulator because the simulator process does not have to be restarted for every simulation and less file activity is required.

Syntax: Non-Piped 2G.6, 3E2, and HPSPICE Simulations

The command formats for non-piped simulations are shown next:

UCB SPICE 2G.6

```
ucbspice2g6 rawfile
```

where:

rawfile is the output binary data file.

The input deck file containing the circuit description and analysis commands comes from

standard input and the output text file containing the results of the simulation goes to *standard output*.

UCB SPICE 3E2

```
spice3e2 -b -r rawfile -o textfile deckfile
```

where:

-b specifies batch mode.

rawfile is the output binary data file.

textfile is the output text file containing the results of the simulation.

deckfile is the input deck file containing the circuit description and analysis commands.

HPSPICE

```
spice2.4n1 deckfile textfile rawfile
```

where:

deckfile is the input deck file containing the circuit description and analysis commands.

textfile is the output text file containing the results of the simulation.

rawfile is the output binary data file.

Syntax: Piped 2G.6, 3E2, and HPSPICE Simulations

The command formats for piped simulations are shown next:

UCB SPICE 2G.6

```
ucbspice2g6 -
```

where:

The "-" denotes that the binary data output is going to the *standard output* pipe. The input deck information comes from the *standard input* pipe and the output text is sent to the file */dev/null.*

UCB SPICE 3E2

```
spice3e2 -s
```

where:

The *-s* option specifies that the input deck information is coming from *standard input* and the binary data output is going to *standard output.*

HPSPICE

```
spice2.4n1 - /dev/null -
```

where:

The first "-" denotes that the input deck information is coming from the *standard input* pipe.

The output text is sent to the file */dev/null.*

The last "-" denotes that the binary data output is going to the *standard output* pipe.

Output Data Formats

The example in the following figure shows the output data format of the *spice2* template simulator supported in IC-CAP.

Output File Format Used For spice2

```
Record 1:  Title card (80 bytes), date (8 bytes), time (8 bytes) TOTAL-96 BYTES
Record 2:  Number of output variables (including "sweep" variable) (2 bytes)
Record 3:  Integer '4' (2 bytes)
Record 4:  Names of each output variable (8 bytes each)
Record 5:  Type of each output (2 bytes each)
          0  = no type
          1  = time
          2  = frequency
          3  = voltage
          4  = current
          5  = output noise
          6  = input noise
          7  = HD2
          8  = HD3
          9  = DIM2
         10  = SIM2
         11  = DIM3
             Outputs 7 through 11 are distortion outputs.
Record 6: The location of each variable within each sweep point. (2-bytes each)
          (Normally just 1,2,3,4,...but needed if outputs are mixed up)
Record 6a: 24 characters that are the plot sub-title if Record 3 is a '4'.
Record 7: Outputs at first sweep point
Record 8: Outputs at second sweep point
Record 9:
          .
          .
          .
last record
All real data are 8-byte quantities.
All complex data are single precision reals, that is 4-byte quantities.
   (4-byte quantity for the real part,
```

```
    4-byte quantity for the imaginary part)
EOF      A special "end-of-file" indicator: 9.87654321D+27 for real data
                                    (9.876E+0,5.432E+0) for complex data
EOI      A 4 byte integer zero indicates the end of all raw data
```

The binary format output by the *spice3* template simulator is shown in the following figure.

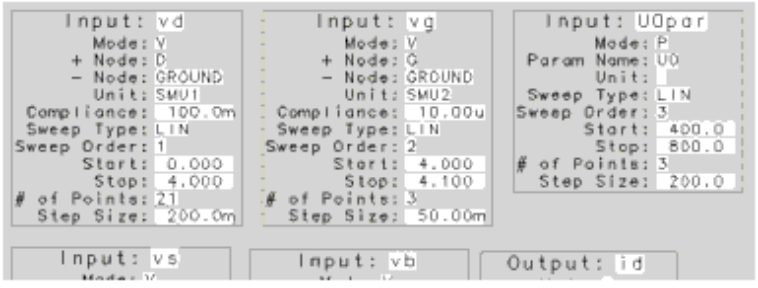
Output File Format Used For spice3

```
Title Card              (Newline (\n) terminated string)
Date and Time           (Newline (\n) terminated string)
Plot Title              (Newline (\n) terminated string)
Flags                   (Newline (\n) terminated string)
Number of Variables     (No. Variables: [an integer])
Number of Points        (No. Points: [an integer])
Version                 (Newline (\n) terminated string)
Variables List
                (Variables:
   [tab]     (index)     [tab]     (name)     [tab]     (type)
   .
                                    .
                                    .
                    {  repeated num_var times }
 where:  index = variable index  [integer]
         name = variable name   [string]
         type = variable type    [string (that is, "current" or "voltage")]
         num_var = number of variables
Binary:                      ( Newline (\n) terminated string indicating the
                              start of the binary data )
Each data point is listed in the order listed in the variables list.
Each real data point is represented by 8 bytes.
Each complex data point consists of the real part and the imaginary
part of 8 bytes each.
There are no separators between data points.
```

SPICE Parameter Sweeps

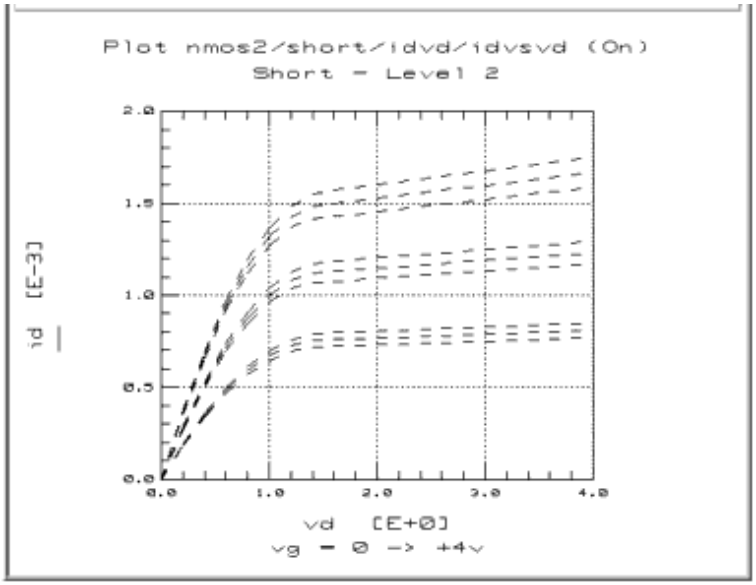
> **Note**
> Parameter sweeps should always be the outer most sweep.
>
> LSYNC sweeps should have master sweeps that are also parameter sweeps.

For SPICE-type simulators, specifying parameter sweeps for devices and circuits requires an input added to the setup (in this example, *nmos2/short/idvd*) with Mode *P*.



SPICE Parameter Sweep Setup Example

IC-CAP performs a simulation for each value of the parameter sweep. The following figure shows the resulting plot.



SPICE Parameter Sweep Plot Example

For additional information on sweeping parameters, refer to the section, *Specifying Parameter or Variable Sweeps* (simulation).

Circuit Model Descriptions

The circuit description for the HSPICE and ELDO simulators is similar to the UCB SPICE simulator circuit description. The details in the following sections also apply to HSPICE and ELDO.

Specifying Simulator Options

For information on available options and their syntax, refer to *Simulation* (simulation) for that simulator. Simulator options are specified in the first line of the circuit definition using the following syntax:

```
.OPTIONS OPT1 = OPTVAL1 OPT2 = OPTVAL2 ... OPTN = OPTVALN
```

where

*OPT*s denote the option keywords used by the simulator

*OPTVAL*s are the corresponding option values. Some options do not require a value; this field may or may not be specified, depending on the option.

A space is the only delimiter required between options.

The nominal and operating temperatures, TNOM and TEMP, are commonly used options; they can also be specified by entering a value (in °C) for the global variables TNOM and TEMP. To do this, enter the variable and its value in the System Variables table in the Utilities application.

- TNOM is the temperature at which the model parameters are extracted; TEMP is the temperature at which the simulation is performed. When performing an optimization to extract model parameters, TEMP and TNOM should be set to the same value so that simulations during optimization are performed at TNOM. TNOM must be defined to guarantee consistency between simulation and extraction.
- In general, TNOM and TEMP can be in any variable table, allowing different Models, DUTs or Setups to use different nominal and operating temperatures. IC-CAP checks for these global variables before running a simulation. If the variable is not found, the value of the option set in the .OPTIONS statement in the Circuit Editor is used when it exists. Otherwise, the circuit is analyzed using the simulator's default values.

IC-CAP automatically adds the option *POST=1* to the options list when the selected simulator is hspice. Specifying this option causes hspice to return the binary raw data file, which IC-CAP requires for reading back the simulated data. This option is not necessary when performing a Manual Simulation from the Simulation Debugger command menu because the data is not read back into IC-CAP.

Describing the Device Model

A device model is used to characterize a single SPICE element of any type. This description requires 2 parts:

- An element statement that calls a defined model
- A *.MODEL* definition, which is identical to a *.MODEL* card in SPICE

The general form of the element statement that calls the device model is:

```
DNAME NNUM1 = NNAME1 NNUM2 = NNAME2 ...NNUMN = NNAMEN MNAME + DPAR1 = DVAL1
DPAR2 = DVAL2 ...DPARN = DVALN
```

where

DNAME is the device name with the first letter being a simulator defined key letter denoting the type of model being specified.

NNUM denotes the node number connections.

NNAME denotes node names corresponding to the node numbers.

DPAR is a predefined DUT parameter name.

DVAL is the specified DUT parameter value. Refer to the SPICE Reference manual for DUT parameter names available for each model.

MNAME is the model name being referenced. This is the same *MNAME* specified

in the *.MODEL* definition described below

A *.MODEL* definition specifies the parameters of a device model that describe a particular element. When a parameter is not specified, the default value in the model is used. The general form of the *.MODEL* definition is:

```
.MODEL MNAME TYPE PNAME1=PVAL1 PNAME2=PVAL2 ...PNAMEN=PVALN
```

where

MNAME is the model name. Regardless of the model name entered in the MNAME field of the .MODEL definition statement, IC-CAP substitutes the name of the Model as it is called in the Model List when the simulator input deck is built.

TYPE is a valid SPICE component type

PNAME is a parameter name for the particular model type

PVAL is the parameter value

As in SPICE, a plus sign (+) that appears as the first character of a line denotes a continuation of the previous line. This continuation character is often used for easier readability when specifying the .MODEL card.

Note
When using the SPECTRE simulator with either the OSI, SPECTRE442, or SPECTRE443 interfaces (see *SPECTRE Interfaces* (simulation)), the LEVEL parameter for a MOS *.MODEL* card may not translate properly. IC-CAP outputs the value as a real number in the netlist, but SPECTRE requires an integer. To work around this issue, use the model type BSIM3 instead of MOS and omit the LEVEL parameter. Alternatively, enclose the LEVEL parameter with parentheses, for example, LEVEL = (11). By doing the later, IC-CAP does not flag it as a model parameter and leaves the expression alone when passing the netlist to SPECTRE.

Describing Subcircuits

A subcircuit model is used to describe a circuit that contains more than 1 element.

The syntax is similar to the syntax in SPICE. The subcircuit description must begin with a *.SUBCKT* and end with a *.ENDS* declaration. Statements between these 2 declarations describe the subcircuit components.

The general form of the first line of a subcircuit definition is:

```
.SUBCKT SUBNAME NNUM1 = NNAME1 NNUM2 = NNAME2 ...NNUMN = NNAMEN + (PAR1=PARVAL1
PAR2=PARVAL2 ...PARN=PARVALN)
```

where

SUBNAME is the name you give to the subcircuit. Regardless of the subcircuit name entered in the SUBNAME field of the .SUBCKT definition statement, IC-CAP substitutes the name of the Model being simulated when the simulator input deck is built.

NNUM are the numbers of the external nodes of the subcircuit. These external nodes are used to connect the subcircuit to another circuit. External nodes in the *.SUBCKT* declaration cannot be 0 (ground), but internal nodes can be connected to ground and any external node to ground in a surrounding circuit.

NNAME is a node name assigned to a node number. As in the device model description, IC-CAP allows the option of equating node numbers to node names. If you assign node names, use these names when specifying the Inputs and Outputs in the Setup.

PAR1 ... PARN are subcircuit parameters that can be passed through subcircuit calls. These parameters are added to the DUT parameter table in IC-CAP.

PARVAL1 ... PARVALN are the corresponding parameter values. These subcircuit parameters become DUT parameters and can be modified in the DUT Parameter Editor.

(While the syntax shown here is correct, passed parameters are ignored by IC-CAP.)

The body of the subcircuit model description contains the components of the subcircuit using element and *.MODEL* statements.

Assigning Node Names

IC-CAP allows the option of equating node numbers to node names in circuit descriptions because it is typically easier to refer to a node by a meaningful name rather than a number. If node numbers only are specified, these node numbers must be used when specifying inputs and outputs. Node identities can also be specified with the format *%<name>*. For example:

```
Q1 1=C 2=B 3=E 4=S NPN or Q1 %C %B %E %S NPN
```

Although HSPICE and ELDO allow alphanumeric characters for node names, node numbers must still be associated with node names because IC-CAP parses HSPICE as a SPICE-type simulator.

When using this format, all node names within the circuit or device must be referenced using the *%[nodename]* syntax.

Test Circuits and Hierarchical Simulation

When characterizing a circuit, it is often necessary to add circuitry around a circuit or device to model the actual measurement Setup. IC-CAP provides a Test Circuit Editor to allow modeling of this additional bias circuitry. Select the DUT from the DUT/Setup panel. Click the Test Circuit tab and enter the test circuit description in the same manner you would enter a Circuit Description. The test circuit definition should include a call to the device or subcircuit defined in the Circuit Editor, as well as the additional circuitry needed to model the external parasitics of the measurement Setup.

Note
When you define a test circuit, the DUT parameter table contains the values specified in the test circuit specification. Regardless of the subcircuit name entered in the SUBNAME field of the *.SUBCKT* declaration, IC-CAP uses the name of the DUT being simulated when the simulator input deck is built.

Subcircuit and device model specifications can be called from inside another Model. This enables you to perform *hierarchical simulations* to study a circuit at different levels.

When making reference to another model, the model name must be used as it appears in the IC-CAP Model List. For example, assume you have defined 3 Models, *model1*, *model2*, and *model3*. *model1* has a circuit model description that is a device definition. The circuit model description for *model2* is a subcircuit definition at the gate level that includes a call to *model1* in a device call statement. And, the circuit model description for *model3* is a subcircuit definition that includes a call to *model2* in a subcircuit call statement. When you simulate a Setup in *model3*, IC-CAP traverses the Model hierarchy and uses the circuit model description defined in *model3*, which includes calls to *model1* and *model2*. The syntax for calling a device model is identical to that described in the *Device Model Description* section.

The general form of the device call is:

```
DNAME NNUM1 = NNAME1 NNUM2 = NNAME2 ...NNUMN = NNAMEN MNAME + DPAR1 = DVAL1
DPAR2 = DVAL2 ...DPARN = DVALN
```

Calling a subcircuit specification allows you to insert an entire subcircuit into a circuit as if it were a single component. The call requires a syntax identical to that used in SPICE. The general form of the subcircuit call is:

```
XNAME NNUM1 NNUM2 ...NNUMN SUBNAME (PARVAL1 PARVAL2 ... PARVALN)
```

where

XNAME is the name of the subcircuit call statement. The only requirement for this name is that it must start with the letter X.

NNUM are the node numbers of the calling circuit that connect to the external nodes of the subcircuit. The calling circuit node numbers need not be the same as the external nodes of the subcircuit. The nodes are connected in the order specified. Specify the same number of nodes declared in the subcircuit definition.

SUBNAME is the name of the subcircuit, previously described by a .SUBCKT *definition*. This must have the name of the model as it appears in the Model List if it is in a different model.

PARVAL are subcircuit parameter values. The order in which they are listed in the subcircuit call statement must match the parameters list in the subcircuit definition.

(While the syntax shown here is correct, passed parameters are ignored by IC-CAP.)

Note
When a test circuit is included in the Model, IC-CAP uses the test circuit description as the *top level* circuit definition. The node number connections defined in the test circuit description, not the circuit description, are used as the external nodes. Because of this, any node-number-to-node-name cross-referencing in the circuit description is not used. Only node names equated to node numbers in the test circuit description can be used when specifying Inputs and Outputs in the Setup Editor. When only node numbers are specified in the test circuit description, (that is, they are not equated to node names) these same node numbers must be used in the Input and Output node fields.

.PARAM

.PARAM can be used in SPICE-type simulators such as hspice, eldo and hspicemodeads.

The .PARAM inside a sub-circuit based netlist always appear in the appropriate parameter list.
That means, if they are defined:

in the Circuit, they appear in the Model Parameter List.
in a Test Circuit, they appear in the Device Parameter List.

The general form is:

```
.SUBCKT SUBNAME NNUM1 = NNAME1 NNUM2 = NNAME2 ...NNUMN = NNAMEN + (PAR1=PARVAL1 PAR2=PARVAL2
...PARN=PARVALN)
[.PARAM PNAME1=PVAL1 PNAME2=PVAL2 ...PNAMEN=PVALN]
                                    -
                                    -
                           elementStatements
                                    -
                                    -
.ENDS
```

Note
Note that not all spice simulators support it.
hspice support .PARAMETER/.PARAMETERS

Circuit Description Syntax

This section describes basic syntax rules for creating a circuit description.

SPICE Simulators

Start an input line with * to denote a comment in the circuit model description or in the input file of the simulation debugger. Although some simulators accept # and *, IC-CAP accepts * only. (# is recognized as a preprocessor directive when the simulator input deck is built. Adding a comment using # causes a simulation generated from a DUT or Setup to fail.)

The following table lists the SPICE element component specifications. For information on available options and their syntax, refer to the *SPICE Reference* manual.

The table below lists the semiconductor device specifications. For information on available

options and their syntax, refer to the *SPICE Reference* manual.

SPICE Element Component Specifications

Component	General Form	Example
Resistor	RXXXXXXX N1 N2 VALUE <TC=TC1<TC2>>	R1 1 2 1000 TC=0.001,0.015
Capacitor	CXXXXXXX N+ N- VALUE <IC=INCOND>	COSC 15 2 10U IC=3
Inductor	LXXXXXXX N+ N- VALUE <IC=INCOND>	LSHUNT 3 29 10U IC=15.7m
Mutual Inductor	KXXXXXXX LYYYYYYY LZZZZZZZ VALUE	K43 LAA LBB 0.999
Transmission Line	TXXXXXXX N1 N2 N3 N4 Z0=VALUE <TD=VALUE> + <F=FREQ <NL=NRMLEN>> <IC=V1,I1,V2,I2>	T1 1 0 2 0 Z0=50 TD=10NS
Linear Voltage-Controlled Current Source	GXXXXXXX N+ N- NC+ NC- VALUE	G1 2 0 5 0 0.1M
Linear Voltage-Controlled Voltage Source	EXXXXXXX N+ N- NC+ NC- VALUE	E1 2 3 14 1 2.0
Linear Current-Controlled Current Source	FXXXXXXX N+ N- VNAM VALUE	F1 13 5 VSENS 5
Linear Current-Controlled Voltage Source	HXXXXXXX N+ N- VNAM VALUE	HX 5 17 VZ 0.5K
Independent Voltage Source	VXXXXXXX N+ N- <<DC> DC/TRAN VALUE> + <AC <ACMAG <ACPHASE>>>	VIN 12 0 DC 6
Independent Current Source	IXXXXXXX N+ N- <<DC> DC/TRAN VALUE> + <AC <ACMAG <ACPHASE>>> + SFFM(0 1 10K 5 1K)	ISRC 23 21 AC 0.333 45.0

SPICE Semiconductor Component Specifications

Component	General Form	Example
Junction Diode	DXXXXXXX N1 N2 MNAME + <OFF><IC=VD>	DCLAMP 3 7 DMOD 3.0 IC=0.2
BJT	QXXXXXXX NC NB NE <NS> MNAME + <OFF> <IC=VBE,VCE>	Q2A 11 26 4 20 MOD1
JFET	JXXXXXXX ND NG NS MNAME + <OFF> <IC=VDS,VGS>	J1 7 2 3 JM1 OFF
MOSFET	MXXXXXXX ND NG NS NB MNAME + <L=VAL><W=VAL><AD=VAL><AS=VAL> +<PD=VAL><PS=VAL><NRD=VAL><NRS=VAL> + <OFF> <IC=VDS,VGS,VBS	M1 2 9 3 0 MOD1 L=10U W=5U

HSPICE Simulator

Basic HSPICE syntax rules are the same as SPICE-type simulators. Refer to the *HSPICE User's Manual* for complete syntax and rules.

Note
Before performing HSPICE simulations, specify the HSPICE version name in the System Variable HSPICE_VERSION. If this variable is not specified, IC-CAP will assume the latest version of HSPICE is being used.

ELDO Simulator

Basic ELDO syntax rules are the same as SPICE-type simulators. In addition to the SPICE-type syntax, FAS user-defined models can be defined and instantiated in the IC-CAP Circuit Editor. An FAS model is defined as:

```
amodel name(pin1,pin2..)
.
<model body>
.
endmodel
```

(smodel and fmodel are also accepted).

The above model is instantiated in a circuit as:

```
yxx name [pin:] 1 2 ... [param: par1 = var1 ...] [model: ...]
```

In addition, the parser accepts the following ELDO constructs:

```
.ADDLIB number pathname
#com . . #endcom
```

FIDEL models (oxx p1:typ p2:typ ... mod=modelname) and transfer functions (FNS, FNZ) are not currently supported by the IC-CAP parser. However, the *#echo* keyword can be used to insert these statements into a circuit in the IC-CAP Circuit Editor.

The *#echo* keyword is available in the IC-CAP Circuit Editor for all supported simulators. *#echo* can be used to pass a deck card or command directly through to the simulator without any parsing by IC-CAP. For example, the line

```
#echo <something that the IC-CAP parser doesn't understand>
```

is sent to the simulator as

```
<something that the IC-CAP parser doesn't understand>
```

The following analog model instantiation syntax is supported for HDL-A:

HDL-A user-defined models with the following syntax can also be instantiated in the IC-CAP Circuit Editor.

```
yxx name(xx) [pin:] 1 2 ... [param: par1 = var1 ...]
```
and
```
yxx name(xx) [pin:] 1 2 ... [generic: par1 = var1 ...]
```

Note
Before performing ELDO simulations specify the ELDO version name in the System Variable ELDO_VERSION. If this variable is not specified, IC-CAP will use the version name specified in the environment variable *eldover*, if it exists. If neither ELDO_VERSION or *eldover* are specified, IC-CAP assumes that the latest version of ELDO is being used.

SPICE Simulator Differences

Subtle differences in syntax, behavior, error handling and calculation of data between the simulators must be considered when creating a circuit description.

- SPICE2 simulations will fail if an underscore is used in the Model name. An error message will appear in the output text file generated by the Simulation Debugger:

```
0*ERROR*: MODEL TYPE IS MISSING
```

- SPICE2 simulations will fail if an underscore is used in a test circuit and DUT name because the simulation input deck uses the DUT name as a model name. An error message will appear in the output text file:

```
0*ERROR*: SUBCIRCUIT NODES MISSING
```

- When attempting a SPICE2 or SPICE3 simulation in the BJT model, if the ideal maximum forward beta parameter BF=0 or the transport saturation current parameter IS=0, the simulation will fail without an error message. (Other parameters may yield similar results when set to zero.)
- SPICE3 is the only simulator that supports the UCB GaAs model. Refer to *UCB GaAs MESFET Characterization* (mesfet) for details on the syntax required to simulate this model.
- HPSPICE is the only simulator that supports the Curtice GaAs model. Refer to *Curtice GaAs MESFET Characterization* (mesfet) for details on the syntax required to simulate this model.
- When using HPSPICE to simulate a UC Berkeley MOSFET model, specify the ucb option in the *.OPTIONS* statement of the circuit description:

```
.OPTIONS ucb
```

- When using SPICE3 with the Simulation Debugger to perform an IC-CAP simulation (as opposed to a manual simulation), an output text file with the following message results: *print card ignored since rawfile was produced*. To generate a more informative output text file, perform a manual simulation. The manual simulation results in an output text file that includes the requested output data values.

Using the PSPICE Simulator with IC-CAP

PSPICE is a SPICE-based circuit simulator developed by MicroSim Corporation. PSPICE uses the same basic numeric algorithms as the UCB SPICE2 simulator but claims superior convergence and performance. Using IC-CAP's Open Simulator Interface, a C-language *Translation Module* is provided that makes PSPICE simulation capability available in IC-CAP. This module and instructions for performing PSPICE simulations in IC-CAP are described here. For general information on the Open Simulator Interface, refer to the section *Adding a Simulator* (simulation).

The IC-CAP/PSPICE link uses UCB SPICE2 as the *template* simulator. When performing a PSPICE simulation in IC-CAP, IC-CAP behaves as if it is performing a SPICE2 simulation. Therefore it generates an input deck in SPICE2 format, calls the simulator and reads back a binary raw data file in SPICE2 format. Through the Open Simulator Interface, the call to the simulator is actually calling the executable version of the C-language *Translation Module, pspice.c.* This executable, called *pspice,_translates the SPICE2 input deck to a PSPICE input format, calls PSPICE to perform the simulation, then translates the PSPICE format binary raw data file to SPICE2 format which is read by IC-CAP. The source code file _pspice.c* is located in the *$ICCAP_ROOT/src* directory.

Note
The IC-CAP/PSPICE translation module *pspice.c* has been updated in IC-CAP 5.0 to support the output binary data format of PSPICE 6.3. Only PSPICE versions with the identical output binary data format will work with this translation module. For older PSPICE versions, use the translation module *pspice5_4.c*, also supplied with this release.

To set up PSPICE simulation capability in IC-CAP:

1. Add the *pspice_simulator to the _usersimulators* file in the directory *$ICCAP_ROOT/iccap/lib*, as shown next.

```
pspice spice2 /<your path>/pspice "<host_machine_name>"
CANNOT_PIPE
```

2. The *host_machine_name* is the host computer for the PSPICE simulations. This name can be left blank ("") if PSPICE and IC-CAP are running on the same computer. Since PSPICE does not have the ability to perform piped simulations in IC-CAP, the CANNOT_PIPE flag must always be set, as shown in the above example.
3. Make the following change to the *pspice.c* program to customize it for your environment:
   - In the *main* routine, specify the full pathname of the actual PSPICE simulator on your system.
4. Compile the translation module using the following command: cc -o pspice pspice.c -lm
5. Move the executable file, *pspice* to a permanent location such as *ICCAP_ROOT/bin.* The location must match the path specified in the *usersimulators* file.
6. In IC-CAP, set the SIMULATOR variable to *pspice* or specify *pspice* with the *Select Simulator* command in the IC-CAP Tools Menu.

The following files are generated in your home directory when running PSPICE simulations in IC-CAP:

- *psp.cir* - PSPICE format circuit description deck file translated from the SPICE2 circuit description deck.
- *psp.raw* - PSPICE formatted binary raw data output file generated by a PSPICE simulation.
- *psp.out* - Output print file generated by a PSPICE simulation.

psp.raw and _psp.out_are automatically removed from your home directory after the simulation is completed in IC-CAP.

Note
When using PSPICE, the LM_LICENSE_FILE environment variable must be set. This variable contains the directory path for the license file required by the PSPICE simulator. Refer to the PSPICE Reference Manual for detailed procedures on installing the PSPICE simulator.

Eldo Simulator

Piped and Non-Piped ELDO Simulations

> Prior to IC-CAP 2008 Addon 2, IC-CAP did not support the CAN_PIPE token for ELDO in usersimulators. This token may now be used on local ELDO simulations with AMS-2007.2a. It is not a true piped mode (netlists and raw files are still written to disk), however it does provide substantial performance improvement by using an interactive mode that avoids restarting ELDO for every simulation.

Non-piped ELDO simulations are identical to non-piped SPICE simulations. This type of simulation is performed when the Simulation Debugger is set to ON. This means that ELDO must be restarted for every simulation. If CANNOT_PIPE is specified for ELDO, even when the Simulation Debugger is OFF, it still performs a non-piped simulation.

Syntax: Piped ELDO Simulations

The command format for an ELDO piped simulation is as follows:

```
eldo -inter -mgls_async
```

where:

> *-inter* specifies the ELDO interactive mode. Commands are sent interactively instead of sending the commands in the netlist.
> *-mgls_async* allows asynchronous communication between the MGLS license manager and ELDO.

load deckname and *run* commands are then passed to the running ELDO process.

Syntax: Non-Piped ELDO Simulations

The command format for an ELDO non-piped simulation is as follows:

```
eldo deckfile
```

where

> *deckfile* is the input deck file containing the circuit description and analysis commands. The name of this deckfile is in the form *<circuit_name>.cir*.

The output binary data file is written to a file named *<circuit_name>.spi3*. This output binary data format is similar to the output binary format of the UCB SPICE3 simulator and is generated when you specify the option

```
.option spi3bin
```

Refer to the ELDO User's Manual for more information.

The output text file, is sent to the file named *<circuit_name>.chi* This file is displayed in the Output table of the Simulation Debugger if it is on.

Circuit Model Descriptions

The circuit description for the ELDO simulator is similar to the UCB SPICE simulator circuit description. For details, refer *Circuit Model Descriptions* (simulation).

Circuit Description Syntax

Basic ELDO syntax rules are the same as SPICE-type simulators. In addition to the SPICE-type syntax, FAS user-defined models can be defined and instantiated in the IC-CAP Circuit Editor. An FAS model is defined as:

```
amodel name(pin1,pin2..)
.
<model body>
.
endmodel
```

(smodel and fmodel are also accepted).

The above model is instantiated in a circuit as:

```
yxx name [pin:] 1 2 ... [param: par1 = var1 ...] [model: ...]
```

In addition, the parser accepts the following ELDO constructs:

```
.ADDLIB number pathname
#com . . #endcom
```

FIDEL models (oxx p1:typ p2:typ ... mod=modelname) and transfer functions (FNS, FNZ) are not currently supported by the IC-CAP parser. However, the *#echo* keyword can be used to insert these statements into a circuit in the IC-CAP Circuit Editor.

The *#echo* keyword is available in the IC-CAP Circuit Editor for all supported simulators. *#echo* can be used to pass a deck card or command directly through to the simulator without any parsing by IC-CAP. For example, the line

```
#echo <something that the IC-CAP parser doesn't understand>
```

is sent to the simulator as

```
<something that the IC-CAP parser doesn't understand>
```

The following analog model instantiation syntax is supported for HDL-A:

HDL-A user-defined models with the following syntax can also be instantiated in the IC-CAP Circuit Editor.

```
yxx name(xx) [pin:] 1 2 ... [param: par1 = var1 ...]
```

and

```
yxx name(xx) [pin:] 1 2 ... [generic: par1 = var1 ...]
```

Note
Before performing ELDO simulations specify the ELDO version name in the System Variable ELDO_VERSION. If this variable is not specified, IC-CAP will use the version name specified in the environment variable *eldover*, if it exists. If neither ELDO_VERSION or *eldover* are specified, IC-CAP assumes that the latest version of ELDO is being used.

Saber Simulator

This section describes the details of using the Saber simulator with IC-CAP. For general information on IC-CAP simulation, refer to *Simulation* (simulation).

The Saber simulator, developed by Analogy, Inc., analyzes analog, digital, event-driven analog and mixed-mode systems.

IC-CAP supports these Saber simulator features:

- Vary command for unlimited sweeps and simulation at multiple operating points
- DC Operating Point Analysis used with DC Transfer, AC Frequency, and Transient Analysis
- Options for each type of analysis (these options must be specified in the IC-CAP Variables Table)
- Parameter sweeps
- Alter command (For details, refer to The Alter Command)
- Hierarchical simulation
- Remote simulation

IC-CAP supports other Saber features as follows. (*Limited support* includes workarounds to achieve desired results that may not be in an ideal format.)

- MAST capabilities.
  - Limited support for the syntax required for model and element development. This can be done in a separate file and included in the Circuit Description using the MAST syntax:
    `<filename`
    where filename is the name of the file that contains the template description of the model or element under development.
  - IC-CAP does not support stimulus conversion to collect data on non-electrical nodes
- Limited support for noise analysis, Fourier analysis, distortion analysis, mixed-mode simulation, and mixed technology simulation. This includes simulations involving non-electrical types such as pressure, revolutions per minute, and torque.

IC-CAP does not support digital state type stimulus and response for mixed-mode simulation. Hypermodels must be used to convert digital states to analog signals.

The Saber simulator supports the following analysis types:

- DC
- AC
- Transient
- Capacitance Voltage (CV)
- 2-Port (S,H,Y,Z,K,A parameter)
- Time-Domain Reflectometry (TDR)

Saber Simulation Example

The circuit description is predefined for all IC-CAP configuration files. Enter this description if a new model is being defined; edit the description to fit specific needs. The syntax is identical to the syntax used for describing circuits in a typical Saber simulation deck.

This simulation example will use the IC-CAP supplied Model sabernpn.mdl.

1. Select the simulator by choosing **Tools > Options > Select Simulator > saber**. Choose **OK**.
2. Choose **File > Open > sabernpn.mdl**. Choose **OK**.
3. View the circuit description by clicking the **Circuit** tab.
   The circuit description is shown in MAST Circuit Description Deck for an NPN Bipolar Transistor. This deck describes the circuit (in this case, a single device) to be used in the simulation.
4. To view the input and output for the fearly setup, click the **DUTs-Setups** tab and select **fearly**;
   The Measure/Simulate folder appears with the inputs vb, vc, ve, and vs, and the output ic. The vc input specifies a voltage source at node C that sweeps linearly from 0 to 5V in 21 steps. The ic output specifies that current at node C be monitored.
   In the Plots folder, icvsvc is specified so that the results of the simulation can be viewed graphically.
5. To simulate, click the **Simulate** button in the Measure/Simulate folder. The Status line displays Simulate in progress.
   When the simulation is complete, the Status line displays IC-CAP Ready.
6. To view the results of the simulation, display the Plots folder and click **Display Plot**. The plot displays measured data represented by solid lines and simulated data represented by dashed lines.

> **Note**
> For syntax examples of running a remote simulation, refer to *Remote Simulation Examples* (simulation).

MAST Circuit Description Deck for an NPN Bipolar Transistor

```
# Saber NPN Device
q..model sabernpn= (IS=le-16,
TYPE= n,
BF = 100,
NF  = 1,
VAF = 1000,
IKF = 10,
ISE = 0,
NE = 1.5,
BR = 1,
NR = 1,
VAR = 1000,
IKR = 10,
ISC = 0,
NC = 2,
RB = 0,
IRB = 10,
RBM = 0,
RE = 0,
RC = 0,
XTB = 0,
```

```
EG = 1.110,
XTI = 3.000,
CJE = 0,
VJE = 750m,
MJE = 333m,
TF = 0,
XTF = 0,
VTF = 1000,
ITF = 0,
CJC = 0,
VJC = 750m,
MJC = 333m,
XCJC = 1.0,
TR = 0,
CJS = 0,
VJS = 750m,
MJS = 0,
FC = 500.0m)
q.qckt C B E S= model = sabernpn, AREA = 1.0
```

Piped and Non-Piped Saber Simulations

Non-piped Saber simulations are identical to non-piped SPICE simulations. However, there are differences between the 2 types of piped simulation. A piped simulation in Saber does the following:

1. Read the input deck from a file upon start up of the simulator.
2. Read in the analysis commands from a pipe connected to *standard input*.
3. Perform the simulation.
4. Send the text output to a pipe connected to *standard output*.
5. Save the output data to files.

Saber is restarted if any topological changes are made to the circuit description. If changes are made which do not affect the topology of the circuit, such as changed parameter values, then alter commands are used and the simulator is not restarted.

Note
The path of the AIM shell interpreter (aimsh) must be specified in usersimulators. IC-CAP uses this utility from the saber installation to interpret the simulation results and read them into IC-CAP. (AIM is a high-level, embedded scripting language that controls and manages user input and other kinds of analyses and processes in SaberDesigner applications.) The default saber specification in $ICCAP_ROOT\iccap\lib\usersimulators is as follows:

```
saber saber $SABER_HOME/bin/saber "" CAN_PIPE "" $SABER_HOME/bin/aimsh
```

Therefore, no modifications to usersimulators are required if SABER_HOME is properly set in your environment before launching IC-CAP.

Syntax: Non-Piped simulations

This section describes the argument syntax required to invoke the template simulator. This information is needed when writing the user translation module, since these are the arguments supplied by IC-CAP when it calls the translation module. For information on the translation module and adding a simulator, refer to *Adding a Simulator* (simulation).

The command format for a Saber non-piped simulation is as follows:

```
saber -b deckfile
```

where:

-b specifies batch mode.

deckfile is the input file name. Saber will read *deckfile* as the input deck file containing the circuit description and *deckfile* as the command file containing the analysis statements.

The textfile is written to a file called *deckfile.*out.

The rawfile information is written to 2 files, called the control file and the data file. The control file is named *deckfile.p1*. *suffix* and the data file is named *deckfile.p2*. *suffix* where *suffix* is a keyword assigned by Saber according to the analysis being performed. Refer to the Saber User's Manual for more information.

Syntax: Piped simulations

The command format for a Saber piped simulation is as follows:

```
saber -c deckfile
```

where:

-c specifies the Saber command mode.

deckfile is the input deck file containing the circuit description.

Saber reads the analysis commands through *standard input.*

The textfile is written to a file called *<deckfile>.out.*

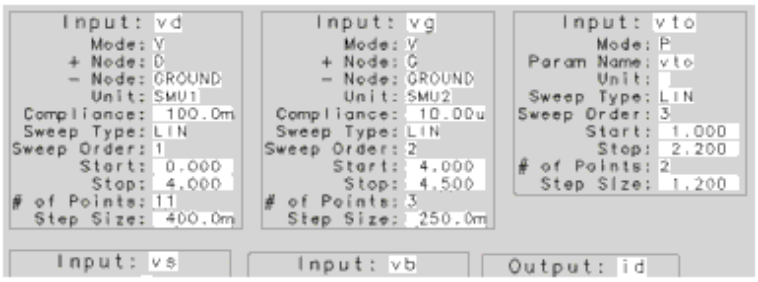
The rawfile is written to a file called *<deckfile>.p1.<suffix>* where *suffix* is a keyword assigned by Saber according to the analysis being performed. Refer to the *Saber User's Manual* for more information.

Saber Parameter Sweeps

Note
The LSYNC sweep is not supported with the Saber simulator.

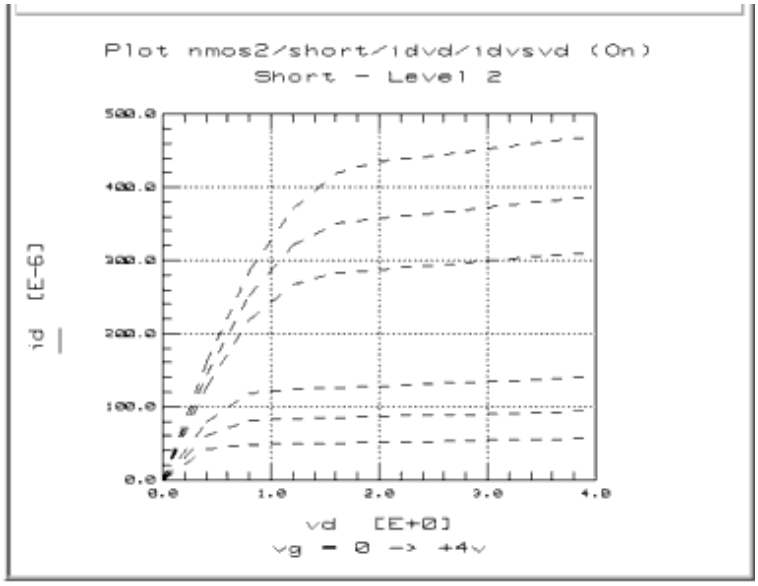
When using the Saber simulator, IC-CAP allows parameter sweeps of only parameters and Saber global variables, such as the global variable for temperature called TEMP. Like SPICE-type simulators, specifying parameter sweeps for devices and circuits is done the same way. Parameter names must be entered in the Name field of the Input table exactly as they appear in the Parameters table. An input for vto, with Mode set to P, is added to the *nmos2/short/idvd* setup, as shown in the following figure.

Saber Parameter Sweep Setup Example



The following figure shows the resulting plot.

Saber Parameter Sweep Plot Example



For additional information on sweeping parameters, refer to *Specifying Parameter or Variable Sweeps* (simulation).

The following sections of this section describe in more detail each of the steps in these simulation overview examples.

The Alter Command

An alter command temporarily changes the value of any element or parameter in a MAST template. It is used to make a change in a template description so that a simulation can be re-executed without reloading the original circuit. The alter command cannot be used to make a change that modifies the topology of a design.

Alter commands are used in IC-CAP Saber simulations when the Circuit Description and Setup information, other than the sweep limits, remain unchanged from the previous simulation.

If only parameter values in the Device Parameters table or Model Parameters table are changed, IC-CAP will not restart the Saber simulator and reload the circuit. Instead, IC-CAP generates alter commands for every parameter, then re-executes the simulation commands. The *USE_ALTER* variable can be specified and set to No to override this

behavior. In this case, Saber is restarted with every simulation whether or not the Circuit Description or Setup was changed. If the *USE_ALTER* variable does not exist, IC-CAP behaves as if the variable were set to Yes.

After a successful simulation, if a resistor is changed from a non-zero to zero value, Saber collapses the nodes. This causes an *implicit* topological change in the circuit that is not recognized by IC-CAP since the Circuit Description or Setup information has not been changed. Turn the *USE_ALTER* variable off by setting it to NO to allow IC-CAP to restart the Saber simulator and reload the altered circuit.

Circuit Model Description

This section discusses the circuit description for the Saber simulator.

Selecting Simulator Options

Saber simulation options are not specified in the circuit description, but rather in the analysis command line. Saber simulator options are set using the *SABER_OPTIONS* variable in the Setup, DUT or System variable tables.

Enter the options in the Value section of the variable exactly as they should appear in the Saber command string. For example, to perform a transient simulation from 0 to 0.8 nsec in 10 psec steps, the Saber command generated in IC-CAP is:

```
tr(ts 1e-11, te 8e-10, tb 0)
```

To specify that all step sizes be fixed instead of variable, append the following option to the Saber command:

```
steps fix
```

To do this in IC-CAP, specify the options command steps fix in the value field of the *SABER_OPTIONS* variable. Simulation now performs the following transient analysis command:

```
tr(ts 1e-11, te 8e-10, tb 0, steps fix)
```

The *SABER_OPTIONS* variable can be specified in a variable table at any level. However, it is important to note that a *SABER_OPTIONS* variable specified in the DUT, Model or System variable tables is used by all simulations executed below that level. For example, if a *SABER_OPTIONS* variable is specified in the DUT variable table, every Setup under that DUT will use the specified option. This may result in simulation errors because 1 particular option may not be valid for every type of analysis being specified in the DUT.

Any number of options can be specified in the *SABER_OPTIONS* variable; they must be separated by a comma.

A Saber analysis in IC-CAP is always preceded by a DC operating point analysis. This DC command can also contain options and can be specified using the *SABER_DC_OPTIONS* variable.

Refer to Saber manuals for available options and corresponding syntax for each simulation type. Invalid options entered into the *SABER_DC_OPTIONS* and *SABER_OPTIONS* variables cause the simulation to fail.

Entering Circuit Descriptions

Circuit descriptions contain templates of devices and components, as well as node connections and model descriptions written in the MAST modeling language. All model parameter names must be specified when defining models. Circuit descriptions can also be read into the IC-CAP Circuit Editor from a file that already contains a description. You must enter circuit descriptions using valid model names and valid parameter names for the particular model being used.

Enter circuit descriptions for a Saber input deck with the Circuit Editor. IC-CAP contains a parser for descriptions written in the MAST modeling language.

There are 2 types of Saber circuit editor descriptions: devices and templates. Syntax rules for each type are described in the following sections.

Device Model Descriptions

A device model is used to characterize a single element of any type. This element can be predefined in the Saber library or defined by the user using the MAST modeling language.

A device model description requires a model definition written in the MAST modeling language and an element statement that calls a defined model.

A model description specifies the values of a device model that describes a particular element. When a parameter is not specified, the default value in the model template is used and the parameter does not appear in the IC-CAP Parameters table. The general form of the model definition is:

```
ENAME..model MNAME = (PNAME1=PVAL1, PNAME2=PVAL2, ...PNAMEX=PVALX)
```

where

ENAME is the name of the element template

MNAME is the user-specified name of the model being defined

PNAME is a parameter name for the particular model type

PVALs are the corresponding parameter values

The general form of the element statement that calls the device model is:

```
ENAME.DNAME NNAME1 NNAME2 ...NNAMEN = model = MNAME, DPAR1 = DVAL1, DPAR2 =
DVAL2 ...DPARN = DVALN
```

where

ENAME is the element template name

DNAME is the device name

NNAME specifies a node name

MNAME is the name of the model being referenced

DPAR is a predefined DUT parameter name

DVAL is the corresponding DUT parameter value

A sample element statement in the MAST modeling language is:

```
q.qckt C B E S = model = sabernpn,AREA = 1.0
```

where

q is the element template name defined in the Saber component library

qckt is the user-specified device name

C, *B*, *E*, and *S* are the node names

sabernpn is the model name. The model corresponding to this model name must be defined in the circuit description before the reference is made.

AREA is a DUT parameter of this model with an assigned value of *1.0*

As in Saber, a line ending with a comma is continued on the next line.

Template Descriptions

A template is used to characterize a circuit that contains more than 1 device. The syntax for defining a template is identical to that of the MAST modeling language. A template can be defined as either an element template or a model template. The general form of the first line of a template element definition is:

```
element template TEMPNAME NNAME1 NNAME2 ...NNAMEN = PAR1, PAR2, ...PARN
```

where

TEMPNAME is the template name

NNAME is a node name of the external node of the template. External nodes are used to connect the template to another circuit.

PAR is the name of the parameter passed into the template

The general form of the first line of a template model definition is:

```
element template TEMPNAME NNAME1 NNAME2 ...NNAMEN = model
```

where

TEMPNAME is the template name

NNAME is the node name of the external node of the template. External nodes are used to connect the template to another circuit.

The body of a model definition defines the model parameters. For more information on writing templates, refer to the *Saber MAST Reference* manual.

When writing a template for model development within IC-CAP, the recommended procedure is to define the template in an external file and include this file in the IC-CAP circuit description using the MAST nomenclature <*filename*> to include a file. This minimizes the changes to be made in the IC-CAP Circuit Description and thereby increases the rate of model development because changes in the external template file will immediately be recognized in IC-CAP.

Non-Numeric Parameter Values

Saber allows non-numeric values for a number of parameters in predefined templates. The MOS model parameter *type* is one example. This parameter can take on the value of _n for an nmos device and _p for a pmos device.

When a Saber input parameter is in alpha format, it does not appear in the IC-CAP Parameters table but is still present in the input deck and passed to the simulator for analysis.

Node Names

Saber accepts alphanumeric names as well as numbers to represent nodes. There is no limit to the number of characters allowed in a node name (the command line has a limit of 1024 characters).

Test Circuits and Hierarchical Simulations

When characterizing a circuit, it is often necessary to add circuitry around a circuit or device to model the actual measurement Setup. IC-CAP provides a Test Circuit Editor to allow modeling of this additional bias circuitry. Select the DUT from the DUT/Setup panel. Click the Test Circuit tab and enter the test circuit description in the same manner you would enter a Circuit Description. The test circuit definition should include a call to the device or template circuit defined in the Circuit Editor, as well as the additional circuitry needed to model the external parasitics of the measurement Setup.

Note
When you define a test circuit, the DUT Parameters table contains the values specified in the test circuit specification. Regardless of the name entered in the TEMPNAME field of the template definition statement, IC-CAP uses the name of the DUT being simulated when the simulator input deck is built.

Template circuit and device model specifications can be called from inside another Model. This allows you to perform *hierarchical simulations* to study a circuit at different levels. For example, assume you have defined 3 Models, *model1*, *model2*, and *model3*. *Model1* has a circuit model description that is a device definition. The circuit model description for *model2* is a template circuit definition at the gate level that includes a call to *model1* in a device call statement. And, the circuit model description for *model3* is a template circuit definition that includes a call to *model2* in a subcircuit call statement. When you simulate a Setup in *model3*, IC-CAP traverses the Model hierarchy and uses the circuit model description defined in *model3*, which includes calls to *model1* and *model2*. The syntax for calling a device model is identical to that described in the Device Model Specifications section above.

The general form of the device call is:

```
ENAME.DNAME NNAME1 NNAME2 ...NNAMEN = model MNAME, DPAR1 = DPAR1, DPAR2 = DVAL2
...DPARN = DVALN
```

Calling a template specification allows you to insert an entire template into a circuit as if it were a single component. The call requires a syntax identical to that used in the MAST modeling language. The general form of the template element call is:

```
TEMPNAME.TNAME NNAME1 NNAME2 ...NNAMEN = TPAR1 = TPARVAL1, TPAR2 = TPARVAL2,
...TPARN = TPARVALN
```

where

TEMPNAME is the name of the template previously described by a template definition. This template definition could exist in a different Model.

TNAME is the user specified name given to this particular instance of the template described by TEMPNAME.

NNAMEs represent the node names of the calling circuit that connect to the external nodes of the template. The calling circuit's node names need not be the same as the external nodes of the template. The order in which you specify these nodes is the order in which they are connected. The same number of nodes as declared in the template definition must be specified.

TPARs are predefined template parameter names. These parameters are defined in the template definition. *TPARVALs* are the corresponding values of the template parameters.

A hierarchical simulation, in which a template in 1 model references a device defined in a different model, requires the use of a MAST *external* declaration in the template definition. For example, assume a MOS device model (Saber template *m* named *nmos2*), which is called in the body of a circuit template called *inverter* in another model. This *inverter*

template must include the following declaration in order for the *nmos2* device model to be recognized.

```
external m..model nmos2
```

The complete template for the inverter circuit is:

```
template inv A B C D E F
electrical A, B, C, D, E, F
{
external m..model nmos2
m.minv A B C D = model = nmos2, l = 10u, w = 10u
m.mload E F A D = model = nmos2, l = 10u, w = 10u
}
```

The external declaration does not need to be added when a template calls another template.

Refer to the Saber manuals for complete syntax and rules of the MAST modeling language.

Saber Libraries

The Saber library of components and templates includes the SPICE components as well as the components developed by Analogy, Inc. Refer to the Saber manuals for a list of supported simulator components, higher level templates and the required specification formats.

Saber Input Deck Comments

To indicate comments in the Saber simulator input deck, start an input line with the pound symbol (#). This denotes a comment in the circuit model description or in the input file of the Simulation Debugger.

Note
The *SABER_DATA_PATH* environment variable must be set. This variable contains the directory paths for the executable files and libraries required by the Saber simulator. Refer to the *Saber Reference Manual* for installation procedures.