Agilent Technologies

Advanced Design System 2002

Analog/RF User-Defined Models

February 2002

Notice

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

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

Warranty

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

Restricted Rights Legend

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

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

Contents

Chapter 1: Building User-Compiled Analog Models

The Model Development Kit provides a graphical user interface that enables you to create your own circuit element models. Creating a model consists of three main steps:

- Defining the parameters whose values will be entered from the schematic
- Defining the symbol and the number of pins
- Writing the C-code itself

When appropriately coded, these elements can be used in linear, nonlinear (Harmonic Balance), transient, and Circuit Envelope simulations. The ADS Application Extension Language (AEL) provides the coupling between the parameters in the schematic design environment and the simulator. The Model Development Kit manages the AEL code generation completely. It also provides a template of the C code so all you need to do is add the relevant equations. Pin numbers and parameters are provided in easy-to-remember C macros and are automatically determined from the symbol drawing.

Note To use this tool, you must have the appropriate C++ compiler installed on your computer. For details, refer to the installation manual for your platform.

Background

User-defined element models are implemented in ANSI-C code. The user-written code is then compiled and linked with supplied object code to make an executable program. This executable program serves as a replacement for the default circuit simulator program. While the equation and parametric circuit capabilities included in the circuit simulators can be used to effectively alter an element or network response through its parameters, the user-compiled model feature allows you access to state vector voltages that affect the model's response currents and charges.

The analysis code for user-compiled models can be written to influence its response depending on its parameters, stimulus controls, analysis type, and pin voltages. The user-defined code can make use of many built-in element models.

You supply the following information using ANSI-C code and AEL (either written by you or generated by the Model Development Kit interface):

- Definition (element names, number of pins, parameter list)
- Modeling functions (linear/nonlinear/transient/noise)
- Library and palette definitions

Note that although the user model is written in C, a C++ compiler is necessary for linking the entire program. The newly integrated elements can then be used in the same way as the built-in elements. A complete model can be developed, compiled and used in a simulation without exiting the schematic design environment.

Note This chapter shows an example on the PC and uses the back-slash (\) convention for file pathnames. On UNIX systems, the pathnames use the (forward) slash (/).

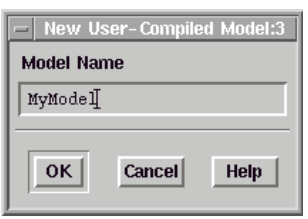
Creating a New Model

This section uses an example to demonstrate the steps that are required to create a new analog user-compiled model using the Advanced Design System's Model Development Kit. The following procedures are shown:

- Starting a new model
- Defining the model parameters
- Creating the model symbol and adding pins
- Writing and compiling the model
- Setting options

Starting a User-Compiled Model

1. Verify that Analog/RF is the current design type.
2. From the Schematic window, choose **Tools** > **User-Compiled Model** > **New User-Compiled Model**.
3. Enter a name (in this example, *MyModel*). Click **OK**.



4. In the dialog box that appears, select the model type:
   - *Linear* can have linear elements only
   - *Nonlinear* can have both linear and nonlinear elements

   Because this example model is a nonlinear diode (with two external pins and one internal node), select **Nonlinear** for the circuit type.

5. Click **OK** and the User-Compiled Circuit Model dialog box appears.

Parameters	-Define input parameters
Model Code	-Create/edit the symbol and its pins -Copy a template for use with new or existing code -Open an existing C code file and edit it -Define various simulation functions -Set compiling options -View compilation status -Save the C code file
Options	-Supply a description for the model -Change the default instance name -Specify which library the model should be stored in -Choose whether or not to include this model in Bill of Materials -Specify whether or not to use an external text editor (and which one) -Specify layout artwork information

Defining the Model Parameters

Use the Parameters tab to define input parameters. Each parameter must be specified by name in the Parameter Name field and a value type (integer, string, real, etc.) assigned to it.

Hint It is usually a good idea (but not required) to specify a default value in the Default Value field.

Optionally you can specify a Parameter Type (unit) and description, as well as a variety of parameter attributes. The Parameter Type controls the scale factor label that is displayed in the dialog box during subsequent editing in the Edit Component

Parameters dialog box. The text you enter in the Parameter Description field is displayed in the component parameter dialog box that appears when you edit a component/instance).

The remaining optional parameter attributes can be used as follows:

- *Display parameter on schematic*—defines whether or not the parameter appears on the schematic
- *Optimizable*—controls whether or not to allow this parameter to be optimized
- *Allow statistical distribution*—controls whether or not this parameter is allowed a statistical distribution when used in conjunction with the simulator's statistical features

---

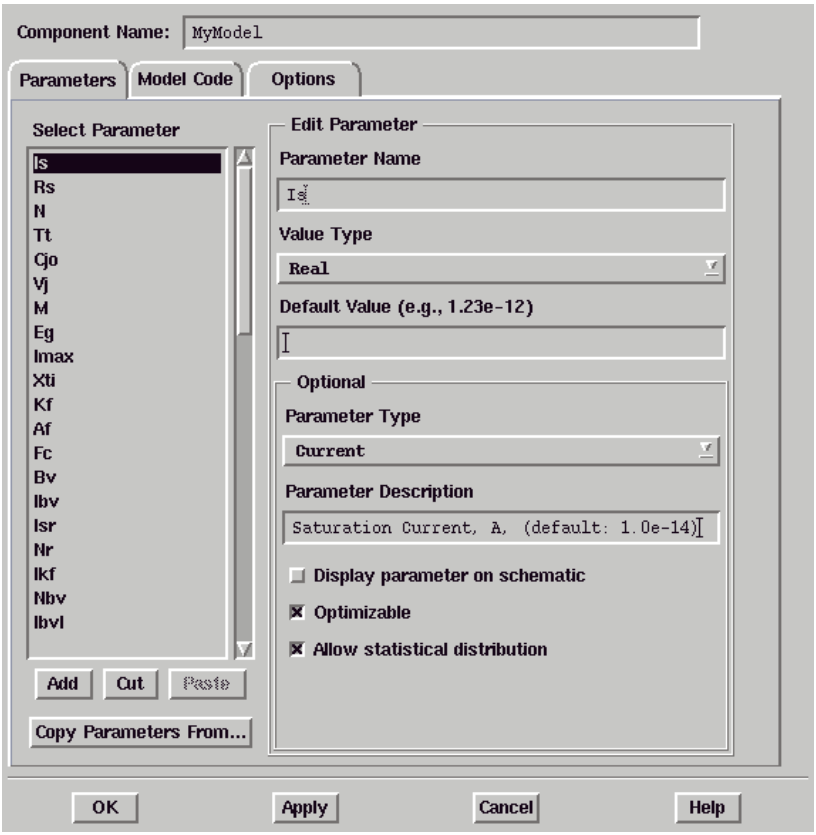
Note Once the model is compiled and inserted in the design, these attributes cannot be modified.

If your parameters are similar to those a supplied component, you can copy and then modify them.

To copy the parameters of a supplied component:

1. Click **Copy Parameters From**.
2. When the library browser appears, select the appropriate library and component, in this example, *Devices-Diodes > Diode_Model (PN-Junction Diode Model)*.
3. The parameter list is updated and can now be modified.

Note Copying parameters from a component will also copy the default symbol, which can then be modified as desired. The symbol can be seen when you click *Create/Edit Symbol and Pins* from the Model Code tab.


Component Name:
MyModel
Parameters
Model Code
Options
Select Parameter
Is
Rs
N
Tt
Cjo
Vj
M
Eg
Imax
Xti
Kf
Af
Fc
Bv
Ibv
Isr
Nr
Ikf
Nbv
Ibvl
Add
Cut
Paste
Copy Parameters From...
Edit Parameter
Parameter Name
Is
Value Type
Real
Default Value (e.g., 1.23e-12)
Optional
Parameter Type
Current
Parameter Description
Saturation Current, A, (default: 1.0e-14)
Display parameter on schematic
Optimizable
Allow statistical distribution
OK
Apply
Cancel
Help

Creating the Model Symbol

Once the parameters are defined, the next step is to generate or create the schematic symbol (or modify the existing one if you used the *Copy Parameters From* option in the Parameters tab). This is done from the Model Code tab.

Click *Create/Edit Symbol and Pins* to switch to symbol view in the Schematic window. If you copied parameters from a supplied component (as in this example), the associated symbol will be there, otherwise a default symbol is generated, based on the number of ports on your design. Add any text or graphics as well as the proper number of external pins. The program will determine how many external nodes exist by the number of pins on the symbol page. You will be able to access the pins in your code via auto-generated macros. You can also modify the symbol graphics and number of pins at any time in the model generation process. Figure 1-1 shows the symbol after modification. The bounding box was deleted, external pins added, and text inserted.

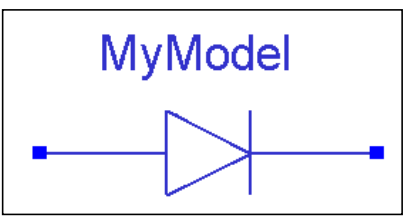


Figure 1-1. Symbol after modification

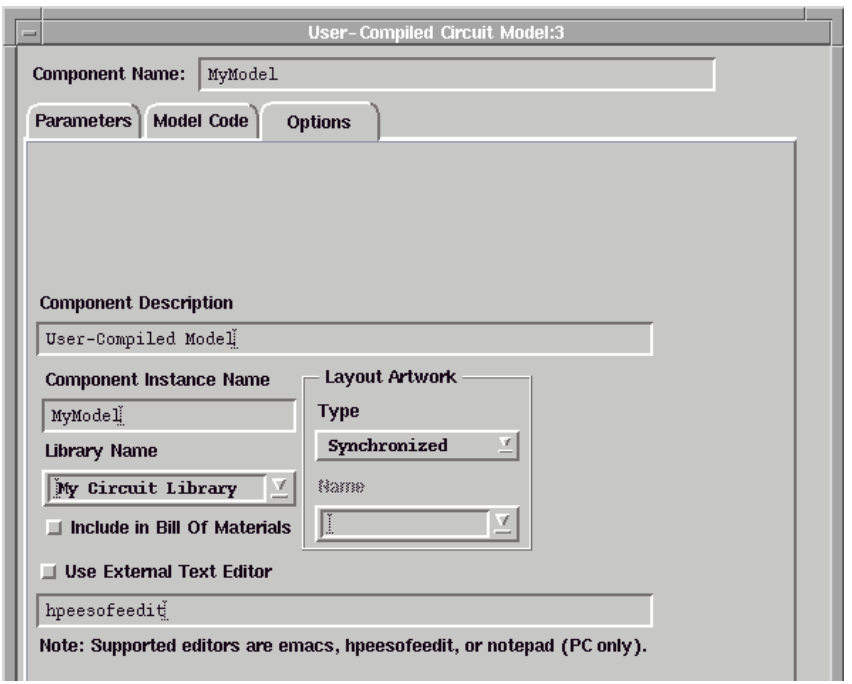
When done defining the symbol, save the file and (optionally) close the window.

Setting Options

Additional options can be set on the Options tab:

- A description for the model
- The library the model should be part of
- Any associated layout artwork and macros
- An attribute to have the component counted in the Bill of Materials
- Specifying an alternate text editor

  By setting an alternate text editor you can use something other than the limited text editor accessed from the dialog box. Note that if you choose to use an external editor, it is up to you to save your files before compiling.

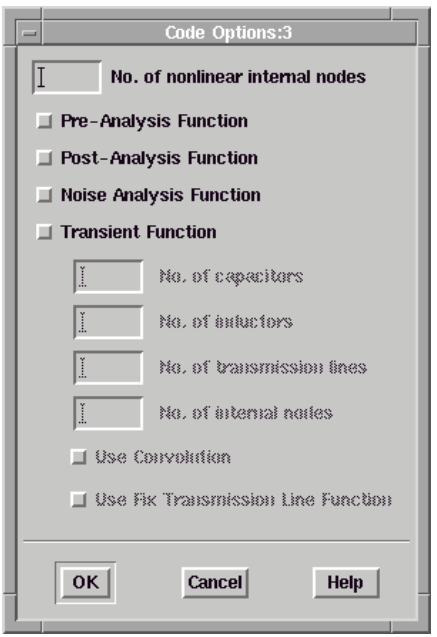


Creating the Code and Compiling the Model

The code generation is done from the Model Code tab.

To set the desired options prior to generating:

Click **Code Options**. In the dialog box that appears, you can select those functions available, as well as set the value for the number of internal, nonlinear nodes. The *Transient* Function provides a field for *No. of internal nodes* for the transient topology. The field appears for both linear and nonlinear models. The *No. of nonlinear internal nodes* field displays only for nonlinear models.



Note The program will generate a file (named *MyModel_h.c* in this example) which has macros defined to either enable or disable the functions. This file is auto-generated for each compile, so you can modify these options if you decide to add another function. The user-edited file is not referenced in the Makefile, it is '#included' from the *MyModel_h.c* file.

A template is provided to assist you in quickly creating your code. It is self-documenting and provides prototypes of all of the functions. (Note that functions are prototyped so that they at least return a value, but this does not guarantee that they will not cause simulation problems.)

To copy the template to your working model file:

1. Click **Create New Code Template**. In this example, the file will be called *MyModel.c*.
2. When the template code appears, edit as necessary to add the correct functionality for your model.
3. If using an external text editor, save the file.

Hint To edit an existing file, click *Open File*.

To set the compiling options prior to compiling:

Click **Compile Options**. Note that the current file is already in the *Object Files to be Linked* list box on the right-hand side. You can optionally select other models to be linked into the simulator. At this time you can also set the following options as desired:

- If compilation is successful, make executable

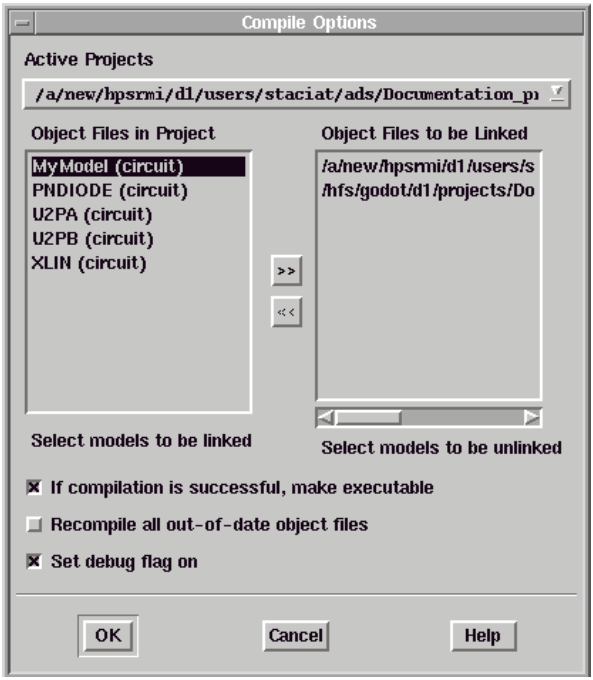
  If the model compiles successfully, this option enables automatic linking. Because the linking can take a few minutes, you may not want to do this all the time.

- Recompile all out-of-date object files

  Verifies whether the other models (not the current model) are out of date, and if so, recompiles them.

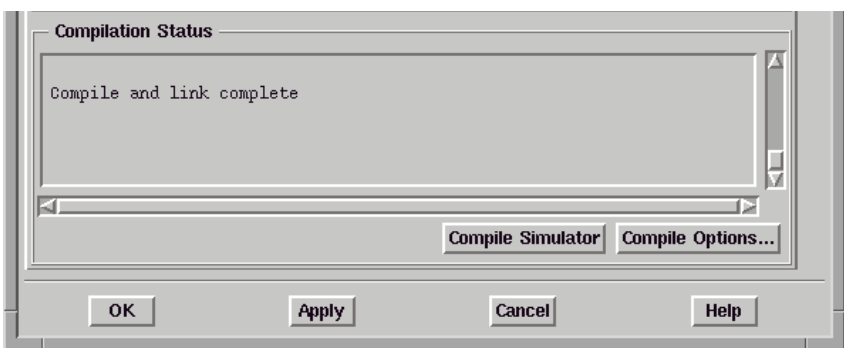
- Set debug flag on

  When this option is selected, the appropriate flag is passed to the compiler to include the debug information in the object code. You will then be able to step through your code when the debugger is invoked. Refer to your debugger documentation for details.

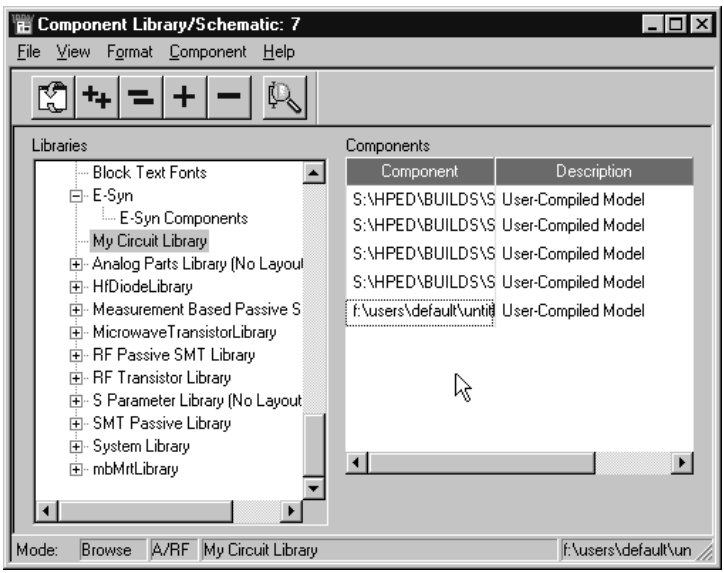


To compile the code:

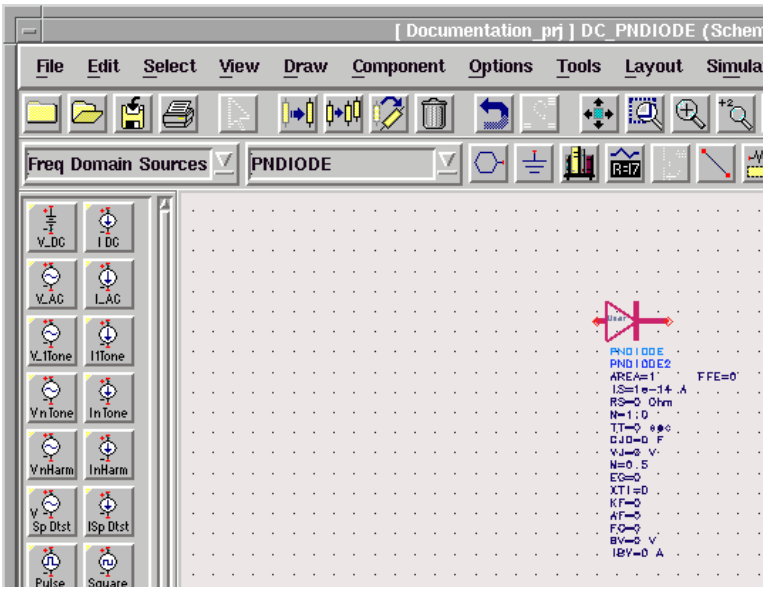
Click **Compile Simulator**. The status is displayed in the Compilation Status pane. If the compile and link process was successful, a new simulator executable will have been generated in the current project directory. (If a prior executable existed, it will have been renamed.)



Once compiled, the model is immediately available in the design environment. From a Schematic window open the Library browser. The model can be found available in the *My Circuit Library* (unless you supplied a unique library name in the Options tab).



Select the component and place it on your schematic, just like any other built-in component.



Characteristics of User-Compiled Elements

An unlimited number of user-defined elements in any number of C modules can be written, compiled and linked to your circuit simulator program. Linear elements can have up to 99 external pins, while nonlinear and transient elements can have any number of external pins and internal nodes.

An element without external pins is treated as a Model Form that has no electrical characteristics. Other elements can refer to this Model Form to obtain parameter values.

Element names and parameter keywords are limited to alphanumeric characters and the underscore character. Names cannot begin with a numeric character. In addition, a leading underscore is not recommended as this can interfere with the built-in variables. Any number of parameters of arbitrary type (integer, real, string or Model Form reference) are allowed for each element. A Model Form reference can refer to either a built-in or a user-defined Model Form.

For use in DC and frequency-domain simulations, an element can have either a linear or nonlinear model. Either type of element can have a transient model for use in a Transient simulation.

Linear and noise analysis responses of elements are computed in the frequency domain. The linear response can be computed either in complex scattering matrix or admittance matrix form. The noise response must be computed in complex current correlation matrix form. A user-defined linear element can call most any existing linear element to obtain its response.

Pre- and post-analysis entry points during program execution are provided to enable such calls and to perform special operations, such as data file reading and memory allocation/de-allocation.

Nonlinear element response is computed in the time domain at a sequence of time samples. Time-to-frequency transformations are computed in the circuit simulator engine and are transparent to the user. Element response is characterized by a set of instantaneous (nonlinear) currents out of each pin, nonlinear charges at each pin and their respective derivatives, all determined by applied pin voltages.

The user's computation functions cannot call other nonlinear elements for their responses. Element models with time-delay dependencies are supported.

Transient element response is computed in the time domain. Element response is characterized by a set of instantaneous (nonlinear) currents out of each pin, nonlinear charges at each pin and their respective derivatives, all as determined by applied pin voltages. Transient computation functions cannot call other elements (except for ideal resistors, capacitors, inductors and transmission lines) for their responses.

Convolution element response can be computed in two ways. One way is to use a linear model frequency response function so that the circuit simulation engine can compute the time-domain impulse response. Alternatively, specific nonlinear transient element response code can be used.

Creating Circuit Elements Interface Definitions and Declarations

Interfacing to the simulator code requires the use of certain ADS defined public C symbols in user-defined element modules. The remainder of this chapter describes the supplied userdefs.h file that contains these symbols (macros, interface data structure typedefs, and function declarations). Note that the Model Development Kit interface will automatically generate most of these functions and that the header file will automatically be included.

Success or failure of a typical interface function call is determined by its return value, 1 for success and 0 for failure. Therefore, the 'boolean' typedef and these macros are provided. Although this boolean type is integer-valued, only TRUE and FALSE values should be associated with it.

```
#define FALSE     0
#define false     0
#define TRUE      1
#define true      1
typedef int boolean;
```

Four macros define the Boltzmann constant (Joules/Kelvin), the charge of an electron (Coulombs), the negative of absolute zero temperature (Celsius), and the standard noise reference temperature (Kelvin). The noise-current correlation parameters returned by an element's noise analysis function must be normalized to FOUR_K_TO--these parameters have admittance dimensions.

```
/* * define some physical constants  */
#define BOLTZ      1.3806226e-23
#define CHARGE     1.6021918e-19
#define CTOK       273.15
#define NOISE_REF_TEMP   2  90.0 /* standard noise reference
          temperature, in Kelvin */
#define FOUR_K_TO   (4.0*BOLTZ*NOISE_REF_TEMP)
/* noise normalization 4kToB, B=1 Hz */
```

This macro obtains the number of items in an array definition at compile time.

```
#define siz(thing) (sizeof(thing)/sizeof(*thing))
```

For clarity, an argument passed by reference can be prefixed by one of these macros in an ANSI function definition and prototype declaration.

```
#define IN    /* input argument to function */
#define OUT   /* output argument: modified/set by function */
#define INOUT /* argument used and modified by function */
#define UNUSED   /* unused argument */
```

Linear response modeled in the frequency domain is complex, so the COMPLEX type is used for admittance (Y), scattering (S), and noise current-correlation parameters.

```
typedef struct
{
  double real;
  double imag;
}
COMPLEX;
```

Each element parameter has a specific type.

```
typedef enum
{
  NO_data = -1,     /* unspecified */
  REAL_data = 0,
  INT_data = 1,
  MTRL_data = 2,   /* for parameter referring to an instance */
  STRG_data = 3,
  COMPLEX_data = 4
}
DataTypeE;
```

Each element parameter definition consists of a keyword string and type.

```
typedef struct {
  char    *keyword;
  DataTypeE    dataType
}
UserParamType;
```

The parameter values of an item are obtained in an array of the UserParamData type. dataType is the discriminator tag to determine the actual value of the union. For example, if it is MTRL_data, value.eeElemInst will refer to a substrate or model form.

```
typedef struct
{
  DataTypeE   dataType;
  union
  {
    double dVal;    /* for REAL_data */
    int iVal;        /* for INT_data */
    void *eeElemInst;   /* for MTRL_data */
    char *strg;       /* for STRG_data */
  }value;
} UserParamData;
```

This type can be used specifically for 2-port elements if the conventional 2-port noise parameters are available.

```
typedef struct
{
  double nFmin;        /* Noise Figure (dB) */
  double magGamma;     /* |opt. source Gamma| */
  double angGamma;      /* <opt. source Gamma (radians) */
  double rnEff;        /* Effective normalized noise resistance */
  double rNorm;       /* Normalizing resistance (ohms) */
} NParType;
```

Each user-element definition is of the `UserElemDef` type. The `pre_analysis` function is useful for one-time operations such as parameter type checking, allocating memory, and reading data files. This routine is called for all types of analysis.

Note that a nonlinear or parametric subnetwork instantiation will be flattened (expanded) in the parent network. If there are two or more uses of a given subnetwork, each occurrence will result in the pre-analysis function (and post-analysis function) being called. The function must be written to properly manage such actions as reading data files and allocating memory.

The `compute_y` function must load the nodal admittance matrix parameters at frequency omega radians/sec into the passed `yPar` array. This function can call `ee_compute_y` (described later) to use another element's admittance parameters.

The `compute_n` function must load the normalized nodal noise current correlation parameters (Siemens, normalized to `FOUR_K_TO`) into the passed nCor array at frequency omega radians/sec and the element admittance parameters, `yPar`. It can call `ee_compute_y` (described later) to make use of another element's admittance and noise correlation matrices.

The `post_analysis` function, called before processing a new circuit, is generally used for cleanup operations such as freeing memory or deleting temporary files. This function is called for all types of analysis. This function is called once for each occurrence of an element, so it must be written to properly manage this situation.

A nonlinear element must contain additional device information in a static area of type `UserNonLinDef` (described later); the pointer `devDef` must point to it.

The `seniorInfo` field is of arbitrary type, and can be used for any extra user-defined data/description that is of no concern to the simulator.

A transient response for an element can be defined in a structure of type `UserTranDef` (described later); the pointer `tranDef` must point to the structure. A transient response function can be defined for either a linear or nonlinear element.

```
typedef struct _UserElemDef UserElemDef;
struct _UserElemDef
{
  char *name;    /* Element name. Not to exceed 8 characters */
  int  numExtNodes; /* Number of external nodes, max. 20 for linear element */
  int  numPars;     /* Number of parameters for this element */
  UserParamType *params;   /* parameter array */


/* pre-analysis function: called after element item parsed successfully */
  boolean (*pre_analysis)(INOUT UserInstDef *pInst);


/* Linear analysis function: called once for each new frequency point.
* Must return the item's admittance matrix in yPar array.
* NULL for nonlinear element    */
  boolean (*compute_y)(IN UserInstDef *pInst, IN double omega, OUT COMPLEX
  *yPar);


/* Linear noise-analysis function: called once for each new frequency point.
Must return the item's noise-current correlation admittance, normalized to
FOUR_K_TO in nCor array.
* NULL if noiseless    */
  boolean (*compute_n)(IN UserInstDef *pInst, IN double omega, IN COMPLEX
  *yPar, OUT COMPLEX *nCor);


/* post-analysis: called before new netlist is parsed */
  boolean (*post_analysis)(INOUT UserInstDef *pInst);


  UserNonLinDef *devDef;  /* User's nonlinear device definition (NULL if
  linear) */
  struct _SeniorType *seniorInfo; /* Senior user defined type and data
  (arbitrary) */
  UserTranDef *tranDef; /* User's transient definition; NULL if none */
};
```

A nonlinear element response is characterized by the functions in the `UserNonLinDef` type.

numIntNodes is the number of nodes internal to the element. In its model, the element code must compute the contributions at all of its pins, which are ordered by external followed by internal pins. If a UserTranDef type is defined for the element, the numIntNodes in that structure must match this definition.

analyze_lin must load only the linear part (complex admittances) of the nonlinear element in the frequency domain. Each admittance must be loaded by calling the primitive add_lin_y function. For a branch admittance between nodes (i, j), 4 calls are needed: +Y for (i, i), (j, j) and -Y for (i, j) and (j, i). analyze_lin can use ee_compute_y to take advantage of pre-existing linear elements. analyze_nl must compute and load the nonlinear response, using the element's pin voltages as input. The passed array pinVoltage contains instantaneous values; however, delayed voltage differences can be obtained using the get_delay_v function.

If P is the total number of pin voltages, formulate non-zero nonlinear current and charge at each pin n as follows:

rn (t) = f (v0 (t), v1(t),..., vP-1(t), vk(t-*k), vl(t-*l),...) where rn is the pin current or charge response,

*k, *l... are ideal delays, independent of the voltages. These responses and their derivatives with respect to voltage (nonlinear conductances, capacitances) must be computed and loaded using the add_nl_iq and add_nl_gc functions, respectively. Note that the derivatives help the simulator to converge to a solution, but do not affect the steady-state nonlinear response—therefore they may work even if not exact. However, under certain simulation conditions in-exact derivatives may cause convergence problems. However, for noise analysis they should be accurate, and for convergence they should be continuous.

In a linear simulation, a nonlinear element must contribute its small-signal linearized response; this is done through the analyze_ac function. The linear part can be loaded by calling the element's analyze_lin function. The linearized part is just the nonlinear conductances and capacitances computed above simply converted to admittances at angular frequency omega and loaded into the circuit matrix using add_lin_y.

Noise contribution of a nonlinear element in a linear simulation is added through the analyze_ac_n function. The linear and linearized noise correlation parameters are loaded using the primitive add_lin_n function. The linearized portion can include shot, flicker, and burst noise contributions.

The modelDef field is of arbitrary type and can be used for any extra user-defined nonlinear model data/description that is of no concern to the simulator.

```
typedef struct _UserNonLinDef UserNonLinDef;
struct _UserNonLinDef
{
  int numIntNodes; /* # internal nodes of device */

/* Evaluate linear part (Y-pars) of device model */
  boolean (*analyze_lin)(IN UserInstDef *pInst, IN double omega)

/* Evaluate nonlinear part of device model:
nonlinear current out of each pin, nonlinear charge at each pin
* derivative (w.r.t. pin voltage) of each
* nonlinear pin current, i.e. nonlinear conductance g,
* derivative (w.r.t. pin voltage) of each
* nonlinear pin charge, i.e. nonlinear capacitance c    */
  boolean (*analyze_nl)(IN UserInstDef *pInst, double *pinVoltage);


/* Evaluate small-signal AC model:
* compute total (linear+linearized) Y-pars of device    */
  boolean (*analyze_ac)(IN UserInstDef *pInst, IN double *pinVoltage, IN
  double omega);
  struct _SeniorModel *modelDef;  /* user-defined Senior MODEL (arbitrary) */

/* Evaluate bias-dependent linear noise model:
compute total (linear+linearized) noise-current correlation parameters
(normalized to FOUR_K_TO, siemens) of device    */
  boolean (*analyze_ac_n)(IN UserInstDef *pInst, IN double *pinVoltage,
  IN double omega);
};
```

A transient response function can be defined for any element (linear or nonlinear) by defining the functions in the `UserTranDef` type shown.

`numIntNodes` is an arbitrary number of nodes internal to the element. In its model, the element must compute the contributions at all its pins, which are ordered and numbered (starting at zero) with the external pins first, followed by internal pins. If a `UserNonLinDef` type is defined for the element, the `numIntNodes` in that structure must match this definition.

Special routines are available to simplify the use of ideal resistors, capacitors, inductors, and transmission lines within a transient element. For the circuit simulator engine to perform the appropriate allocations, the number of these elements (except resistors) must be predefined using `numCaps`, `numInds`, and `numTlines`.

The analyze_tr function must compute and load the instantaneous time domain response of the element, using the element's pin voltages as inputs. The passed array pinVoltage contains the instantaneous voltages at both the external and internal pins.

If P is the total number of pin voltages, formulate nonlinear current and charges at each pin n as follows:

rn (t) = f (v0(t),v1(t),..., vP-1 (t)) where rn is the current out of the pin or charge response. These responses and their voltage derivatives (nonlinear conductances and capacitances) must be computed and loaded using the add_tr_iq and add_tr_gc functions, respectively. Note that the derivatives are used to help converge to a solution, therefore the simulator may reach a solution even if they are not exact. However, under certain simulation conditions, inexact derivatives may cause convergence problems. Also, for convergence reasons, they should be continuous.

The fix_tr function is called just before transient analysis begins. Its only purpose is to set up ideal transmission lines for the user. Using the add_tr_tline function, transmission line nodes and physical constants are defined. Once the transmission line is defined here, time-domain analysis of it is performed automatically without any further action by the user in the analyze_tr function.

```
typedef struct _UserTranDef UserTranDef;
struct _UserTranDef
{
  int     numIntNodes;   /* internal nodes of device */
  int     numCaps;     /* number of explicit capacitors */
  int     numInds;     /* number of explicit inductors */
  int    numTlines;    /* number of explicit transmission lines */
  boolean useConvolution;   /* use linear response for convolution */
/* Evaluate transient model
* nonlinear currents out of each pin,
* nonlinear charge at each pin
* derivative (w.r.t. pin voltage) of each
* nonlinear pin current, i.e. nonlinear conductance g,
* derivative (w.r.t. pin voltage) of each
* nonlinear pin charge, i.e. nonlinear capacitance c   */
  boolean (*analyze_tr)(IN UserInstDef *pInst, IN double *pinVoltage);
/* Pre-transient analysis routine used to allocate, compute and
* connect ideal transmission lines   */
  boolean (*fix_tr)(IN UserInstDef *pInst);
};
```

Each user-defined item placed in a design is represented in the ADS Simulator by the item type UserInstDef. All the fields, except seniorData, in an item are set up by ADS Simulator and must not be changed. seniorData can refer to arbitrary data and is meant to be managed by user code exclusively.

```
typedef struct _UserInstDef UserInstDef;
struct _UserInstDef
{
  char   *tag;   /* item name */
  UserElemDef   *userDef;   /* access to user-element definition */
  UserParamData   *pData;   /* item's parameters */
  void   *eeElemInst;   /* EEsof's element item */
  void   *eeDevInst;   /* EEsof's nonlinear device item */
  void   *seniorData;   /* data allocated/managed/used only by
  Senior module (arbitrary) */
};
```

The get_params function, below, loads the passed item eeElemInst parameter values into pData, which must be big enough to store all parameters. It is used to obtain referenced item (such as model substrate) parameters. Note that user-defined item parameters are already available in the UserInstDef.pData array, so there is no need to call get_params for user item parameters. It returns TRUE if successful, FALSE otherwise.

```
extern boolean get_params (IN void *eeElemInst, OUT UserParamData *pData);
```

These functions are useful to indicate program status in various stages of execution, such as during module boot-up, element analyses, and pre- or post-analysis.

```
extern void send_info_to_scn (IN char *msg); /* write msg to Status/
Progress window */
```

```
extern void send_error_to_scn (IN char *msg); /* write msg to Errors/
Warnings window */
```

In nonlinear analyses, for each set of independent input values (bias, frequency, power, or swept variable), ADS simulator attempts to find the steady state solution iteratively. In each iteration, nonlinear parts of all element items, including user-defined items, are evaluated. This function returns TRUE whenever the first iteration is in progress. It is most useful for parameter range checking, which is sufficient to do at the first iteration.

```
extern boolean first_iteration (void);
```

This function returns TRUE whenever the circuit is being analyzed at the first point in a frequency plan. Note that this can happen many times in one simulation command for example, if there is another swept variable, or if an optimization/yield analysis is requested.

If a one-time-only operation is to be performed per circuit, the pre_analysis function is recommended instead of this function.

```
extern boolean first_frequency (void);
```

The function below computes the normalized complex noise correlation matrix for a passive element, given its Y-pars, operating temperature and number of pins.

```
extern boolean passive_noise (IN COMPLEX *yPar, IN double tempC, IN int
numNodes, OUT COMPLEX *nCor);
```

The function below computes the normalized complex noise correlation 2*2 matrix for an active 3-terminal, 2-port element/network, given its Y-pars and measured noise parameters. Note that if numFloatPins is 2, the common (reference) third terminal is ground.

```
extern boolean active_noise (IN COMPLEX *yPar, IN NParType *nPar, int
numFloatPins, OUT COMPLEX *nCor);
```

The function below must be called (usually from nonlinear model's analyze_lin and analyze_ac procedure) to add the linear complex Y-parameter (iPin, jPin) branch contribution. This call must be done even for linear capacitive branches at DC (omega = 0), this will establish the Jacobian matrix entry location for subsequent non-zero harmonic omega.

```
extern boolean add_lin_y (INOUT UserInstDef *userInst, IN int iPin, IN int
jPin, IN COMPLEX y);
```

The function below must be called (from nonlinear model's analyze_ac_n function) to add the complex noise-current correlation term iNcorr (Siemens, normalized to FOUR_K_TO) from the (iPin, jPin) branch.

```
extern boolean add_lin_n (INOUT UserInstDef *userInst, IN int iPin, IN int
jPin, IN COMPLEX iNcorr);
```

The function below must be called (from nonlinear model's analyze_nl function) to add the nonlinear conductance and capacitance contribution for the (iPin, jPin) branch.

```
extern boolean add_nl_gc (INOUT UserInstDef *userInst, IN int iPin, IN int
jPin, IN double g, IN double c);
```

The function below must be called (from nonlinear model's analyze_nl function) to add the nonlinear current and charge contribution at the device pin iPin.

```
extern boolean add_nl_iq (INOUT UserInstDef *userInst, IN int iPin, IN
double current, IN double charge);
```

The function below can be called (from nonlinear model's analyze_nl function) to get tau seconds delayed (iPin, jPin) voltage difference. Note that tau must not be dependent on device pin voltages--it is an ideal delay.

```
extern boolean get_delay_v (INOUT UserInstDef *userInst, IN int iPin, IN
int jPin, IN double tau, OUT double *vDelay);
```

Any transient support function that follows can use ground as a pin by using this special macro:

```
#define GND -1
```

The function below can be called (from the transient model's analyze_tr function) to obtain the current time value, in seconds, of the transient analysis.

```
extern double get_tr_time (void);
```

The function below must be called (from the transient model's analyze_tr function) to add the nonlinear conductance and capacitance contribution for the (iPin, jPin) branch.

```
extern boolean add_tr_gc (INOUT UserInstDef *userInst, IN int iPin, IN int
jPin, IN double g, IN double c);
```

The function below must be called (from the transient model's analyze_tr function) to add the nonlinear current and charge contribution at the device pin iPin.

```
extern boolean add_tr_iq (INOUT UserInstDef *userInst, IN int iPin, IN
double current, IN double charge);
```

The function below can be called (from the transient model's analyze_tr function) to add a resistor of rval Ohms between pin1 and pin2. The contribution of this resistor need not be included in the other calculated currents, charges and derivatives. If rval is less than 10^{-6}, rval is set equal to 10^{-6}.

```
extern boolean add_tr_resistor (INOUT UserInstDef *userInst, IN int
pin1,IN int pin2, IN double rval);
```

The function below can be called (from the transient model's analyze_tr function) to add a capacitor of cval Farads between pin1 and pin2. The contribution of this capacitor need not be included in the other calculated currents, charges and derivatives. If cval is zero, an open circuit will exist between pin1 and pin2.

```
extern boolean add_tr_capacitor (INOUT UserInstDef *userInst, IN int
pin1,IN int pin2, IN double cval);
```

The function below can be called (from the transient model's analyze_tr function) to add an inductor of lval Henries between pin1 and pin2. The contribution of this inductor need not be included in the other calculated currents, charges and derivatives. If lval is zero, a short circuit will exist between pin1 and pin2.

```
extern boolean add_tr_inductor (INOUT UserInstDef *userInst, IN int pin1,
IN int pin2, IN double lval);
```

The function below can be called (from the transient model's `fix_tr` function) to add an ideal transmission line. The impedance of the line is z0 Ohms and the propagation delay time of the line is td seconds. The loss parameter is used to describe the voltage attenuation on the line; a loss of 1.0 specifies a lossless line; a loss of 0.5 specifies an attenuation of 6 dB. The time domain simulation of this transmission line will be computed automatically with no further action by the user in the `analyze_tr` function.

```
extern boolean add_tr_tline (INOUT UserInstDef *userInst, IN int pin1, IN
int pin2, IN int pin3, IN int pin4, IN double z0, IN double td, IN double
loss);
```

The function below must be called by user-defined element code (possibly from a `pre_analysis` function) that wants to use an element built in ADS. It returns a pointer to an allocated ADS item if successful, NULL otherwise. This pointer must be saved (possibly with the user-defined element item, in its `seniorData` field) and passed to `ee_compute_y` or `ee_compute_n`.

```
extern void *ee_pre_analysis (IN char *elName, IN UserParamData *pData);
```

These functions allow access to Advanced Design System elements for linear and noise analysis. Note that parameter data `pData` must be supplied in SI units, where applicable. They return `TRUE` if successful, `FALSE` otherwise. To determine parameter order, execute the simulator binary (hpeesofsim) using the **-h** flag and the name of the parameter (e.g., *$HPEESOF_DIR/bin/ hpeesofsim -h MLIN*).

```
extern boolean ee_compute_y (INOUT void *eeElemInst, IN UserParamData
*pData, IN double omega, OUT COMPLEX *yPar);

extern boolean ee_compute_n (INOUT void *eeElemInst, IN UserParamData
*pData, IN double omega, IN COMPLEX *yPar,OUT COMPLEX *nCor);
```

The function below must be called by user-defined element code (possibly from a `post_analysis` function) for every `ee_pre_analysis` call to free memory allocated for the Advanced Design System item `eeElemInst`.

```
extern boolean ee_post_analysis (INOUT void *eeElemInst);
```

This returns a pointer to the `UserInstDef` user-defined item if `eeElemInst` is indeed an item of a user-defined element, NULL otherwise.

```
extern UserInstDef *get_user_inst (IN void *eeElemInst);
```

The function converts between S- and Y-parameters. If direction is 0, it computes S-parameters into outPar using inPar as Y-parameters. If direction is 1, it computes Y-parameters into outPar using inPar as S-parameters. rNorm is the S-parameter normalizing impedance in ohms, and size is the matrix size. It returns TRUE if successful, FALSE otherwise.

```
extern boolean s_y_convert (IN COMPLEX *inPar, OUT COMPLEX *outPar, IN int
direction, IN double rNorm, IN int size);
```

Series IV Functions

The following C macros replace corresponding Series IV functions, which returned scale factors to convert a parameter value to SI. In ADS, parameter data are always considered to be in SI; hence these macros always return 1.0, and are meant only for Series IV compatibility.

```
#define get_funit(eeElemInst)  1.0 /* freq unit */
#define get_runit(eeElemInst)  1.0 /* resistance unit */
#define get_gunit(eeElemInst)  1.0 /* conductance */
#define get_lunit(eeElemInst)  1.0 /* inductance */
#define get_cunit(eeElemInst)  1.0 /* capacitance */
#define get_lenunit(eeElemInst)  1.0 /* length unit */
#define get_tunit(eeElemInst)  1.0 /* time unit */
#define get_angunit(eeElemInst)  1.0 /* angle unit */
#define get_curunit(eeElemInst)  1.0 /* current unit */
#define get_volunit(eeElemInst)  1.0 /* voltage unit */
#define get_watt(eeElemInst, power)  (power) /* power unit */
```

Referencing Data Items

A user-defined element parameter can be a reference to an ADS or a User-Defined item. Use the get_params function to obtain the referenced item's parameters.

As an example, if you are creating a microstrip element and need an MSUB reference, the third parameter, for example, can be:

```
{"MSUB", MTRL_data}
```

Then the array entry `userInst->pData[2]` will be such that `pData[2].value.eeElemInst` points to the referred MSUB item in the circuit. The MSUB parameters can then be obtained through a get_params call:

```
get_params(userInst->pData[2].value.eeElemInst, mData)
```

This will copy the MSUB parameters:

Er Relative dielectric constant.

Mur Relative permeability.

H (m) Substrate thickness.

Hu (m) Cover height.

T (m) r Conductor thickness.

Cond (Siemens/m) Conductor conductivity.

TanD Dielectric Loss Tangent.

Rough Conductor surface roughness.

into mData[0...7] locations. The `mData` array must be dimensioned large enough to hold all the referenced item's parameters. If a parameter value is not set or available, the `dataType' enum value will be NO_data.

If the referenced item (using the third parameter again) is an item of a user-defined Data Item, then you can obtain a pointer to the user item as follows:

```
refInst = get_user_inst(userInst->pData[2].value.eeElemInst)
```

The function get_user_inst will return `NULL` if the passed argument is not a user-defined item.

Displaying Error/Warning Messages

You can flag errors within a function in a user-defined element module and send messages to the Simulation/Synthesis panel. You can also write helpful status and debug messages to the Status/Summary panel. The following functions can be used for sending the message to respective locations:

- *extern void send_error_to_scn (char *)*
  writes message to Errors/Warnings panel
- *extern void send_info_to_scn (char *)*
  writes message to Status/Progress panel

The argument of these functions is a character pointer that is the error message string.

Examples:

```
send_error_to_scn("divide-by-zero condition detected");
send_info_to_scn("value of X falls outside its valid range");
```

Using Built-In ADS Linear Elements in User-Defined Elements

A user-defined element can call an ADS linear element to obtain the latter's COMPLEX Y and noise-correlation parameters. However, nonlinear devices, model items, and independent sources cannot be called in a user-defined element module. The relevant functions in the interface to support this feature are described below:

```
extern void *ee_pre_analysis (char *elName, UserParamData *pData);
```

To allocate a pseudo item and do any pre-analysis processing such as data file reading, the user-defined element must first call this function. The second argument, `UserParamData *pData`, must contain the data of correct type and in the order expected by the ADS element. The values must be in SI units where applicable. This function is usually called from the user-defined element's `pre_analysis` function. It returns a pointer to an allocated ADS item if successful, `NULL` otherwise. This pointer must be saved (possibly with the user-defined element item, in its `seniorData` field) and passed to `ee_compute_y` or `ee_compute_n`.

```
extern boolean ee_compute_y (void *eeElemInst, UserParamData *pData,
double omega, COMPLEX *yPar);
```

The function below obtains N×N COMPLEX Y-parameters of the N-node (excluding ground) ADS element item in the user-supplied `yPar` array at frequency omega radians/sec. It returns `TRUE` if successful, `FALSE` otherwise.

```
extern boolean ee_compute_n (void *eeElemInst, UserParamData *pData,
double omega, COMPLEX *yPar, COMPLEX *nCor);
```

The function below obtains the N×N COMPLEX Noise correlation matrix parameters, given *omega* and the N×N COMPLEX Y-pars. It returns `TRUE` if successful, `FALSE` otherwise.

```
extern boolean ee_post_analysis (void *eeElemInst);
```

This function must be called for each `pseudo` item created by each `ee_pre_analysis` call by the user-defined element. This frees up the memory allocated for the ADS item. It is usually called from the user-defined element's `post_analysis` function. It returns `TRUE` if successful, `FALSE` otherwise.

Booting All Elements in a User-Defined Element File

In order to keep the code modular, each user-defined element file can contain at most a single external/public symbol definition; this is the booting function, usually named `boot_abc` for a module named `abc.c`:

```
boolean boot_abc(void)
```

This function is called once per module—at program bootup. If the ModelBuilder interface is used, only one model per file is allowed. However, multiple files can be combined into one larger module by the user. The call to boot the module must be included in the self-documented `userindx.c` file at the appropriate location. The module's file name must be added to the `user.mak` `USER_C_SRCS` definition.

All user-defined elements in the module can be defined in a *static* (with module-scope) `UserElemDef` array, and booted by calling the provided function `load_elements` from `boot_abc`:

```
extern boolean load_elements (UserElemDef *userElem, int numElem);
```

If necessary, you can include code for technology/data file reading, as well as for automatic AEL generation, in the boot function.

Porting Libra Senior to the ADS 1.0 Model Builder Interface

The ADS 1.0 Model Builder interface is designed to allow Series IV-Libra Senior models to be easily ported. Most models will require recompiling using the supplied makefiles. However, several features will require code modification if they were used:

- DataItems
- Default Units
- Substrates
- Calls to built-in elements

Data Items

The Data Item is replaced by references to an AEL expression, or equation. Users can still access default values, however, the '*' value is no longer supported and must be replaced by references to built-in values. For example, the Series IV Data Items can be set via comparable VAR expressions:

_DEFAULT_TEMP = 290.15

_DEFAULT_REF = 50

Instead of using the '*' value to indicate the default, the parameter should be left blank. If the value on the schematic is set to blank the simulator defaults to the Series IV default values:

TEMP	Temperature	TEMP = 27 C
RREF	Reference resistance	R = 50 Ohm
TAND	Dielectric loss tangent	TAND = 0
PERM	Permeability & Mag loss tangent	MUR = 1; TANM = 0
SIGMA	Dielectric conductivity	SIGM = 0

Default Units

The concept of Default Units is not used in ADS. Instead, each parameter on a component can have a scale factor. The simulator's parsing utility replaces the parameter value+scale factor with the proper number (e.g., 1.23GHz is supplied to the user's program as 1.23e9). Any calls to `get_unit` functions are still supported; they merely return a value of 1.

Note that the scale label is ignored:

```
Resistance    = 100 kOhm
              = 100 * 1e3
              = 100 kBricks
              = 100 kHz
```

All calls to any of the units functions are all equivalent (the user's code will receive a value of le5).

Substrates/Built-In Models

The substrate use model has changed considerably in ADS and requires users to modify their calls to any substrate components. Certain parameters have been moved from Data Items to the instance, others to the substrate component. The order and name of the parameters has also changed. Conductivity is specified instead of resistivity.

User calls to built-in models may also have changed, depending on the element. Users will have to reconfirm parameter ordering.

To determine parameter order, execute the simulator binary (hpeesofsim) using the -h flag and the name of the parameter (e.g., $HPEESOF_DIR/bin/ *hpeesofsim -h MLIN*).

AEL Changes

Series IV AEL may require substantial modification. For example, AEL references to Series IV forms (rvopt, etc.) should be changed to 'StdFormSet.' It may be expedient to use the Model Development Kit to generate the AEL file rather than edit a Series IV file.

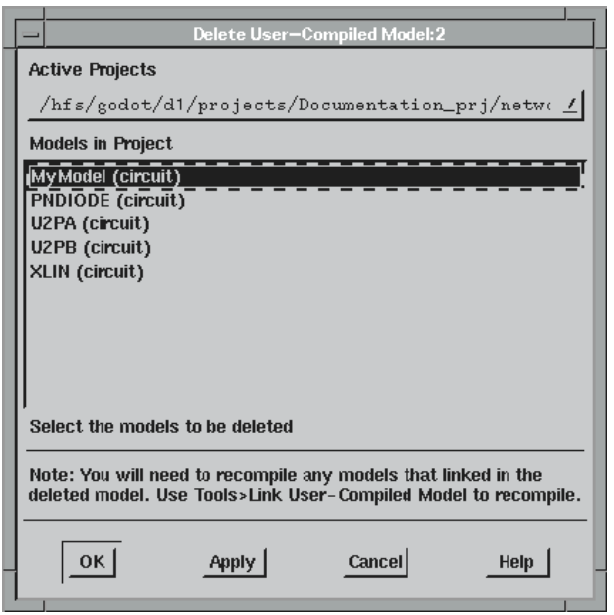
```
create_item("U2PA","User-Compiled Model","U2PA",16,-1,NULL,"Component
Parameters","","%d:%t%#%44?0%:%31?%C%:_net%c%;%;%e%b%r%8?%29?%:%30?%p%:%k%?
[%1i]%;=%p%;%;%;%e%e","U2PA","%t%b%r%38?%:\n%39?all_parm%A%:%30?%s%:%k%?
[%1i]%;=%s%;%;%;%e%e%;","U2PA",3,NULL,64,
create_parm("R1","Pin 1 to GND Resistance ",68608,"StdFormSet",-1,
prm("StdForm","50")),
create_parm("R2","Pin 1 to Pin 2 Resistance",68608,"StdFormSet",-1,
prm("StdForm","50")),
create_parm("R3","Pin 2 to GND Resistance",68608,"StdFormSet",-1,
prm("StdForm","50"))
);
```

Opening an Existing Model

To work on an existing user-compiled model, open the project where the model exists. Use the menu command Tools->Open User-Compiled Model. Select the model from the browser.

Deleting a User-Compiled Model

To permanently delete an existing user-compiled model, open the project where the model exists. Use the menu command Tools->Delete User-Compiled Model. Select the model to be deleted from the list. Click *Apply* to delete the model and leave the dialog box active, or click *OK* to delete the model and dismiss the dialog. Click *Cancel* to dismiss the dialog without deleting the model.

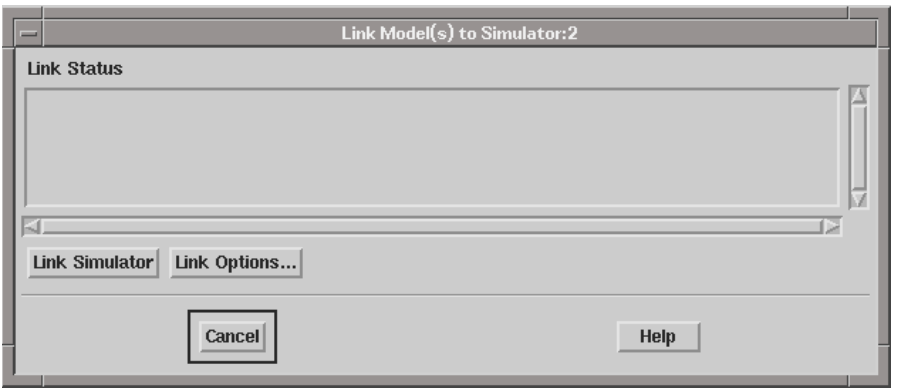


The program prompts you for confirmation before deleting.

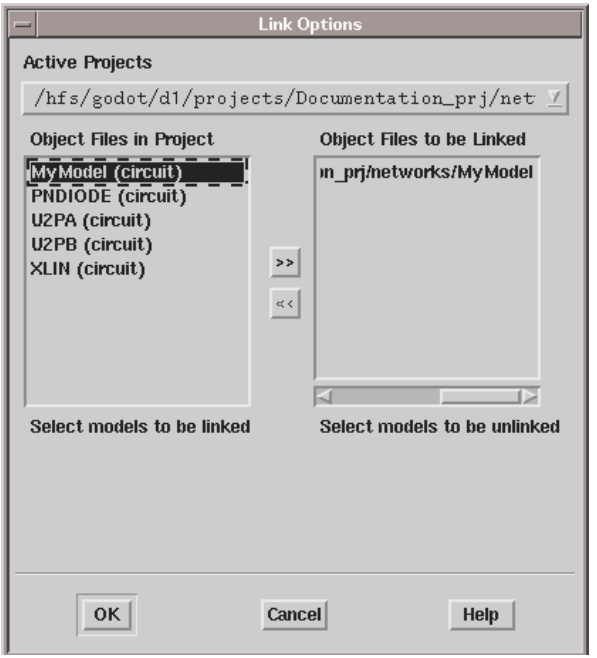
Linking User-Compiled Models

Previously compiled user models can be linked without recompiling. Move the model files into the projects networks directory. Select Tools->Link User-Compiled Model... from the menu.

The Link Model dialog box opens.



Click the *Link Options...* button to open the Link Options dialog box. From this dialog, select the models to be linked from the left hand column (Object files in Project), and click the >> button to move them to the right hand column (*Object Files to be Linked*)



Click the *OK* button to close the dialog. Click the *Link Simulator* button to link the chosen models into a new executable. The new executable will be placed in the projects directory (if an existing executable was already there, it will be renamed).

Managing Model Files

The Analog Model Development Kit interface uses several files as a means of managing the C-code and the compile and link process.

The program copies certain files from the *$HPEESOF_DIR/modelbuilder/lib* directory to the local project directory. For a model file called *MyModel*, when the *Create New Code Template* button is clicked the file *cui_circuit.template* is copied to *MyModel.c*. When the *Compile* button is clicked, *hpeesofdebug.mak* or *hpeesofopt.mak* is always copied (depending on setting of *Debug* check box) and *userindx.c*, *user.mak* are copied if they are not there, and *MyModel_h.c*, *cui_indx.c*, *modelbuilder.mak* are created if they do not already exist, and existing files may be overwritten.

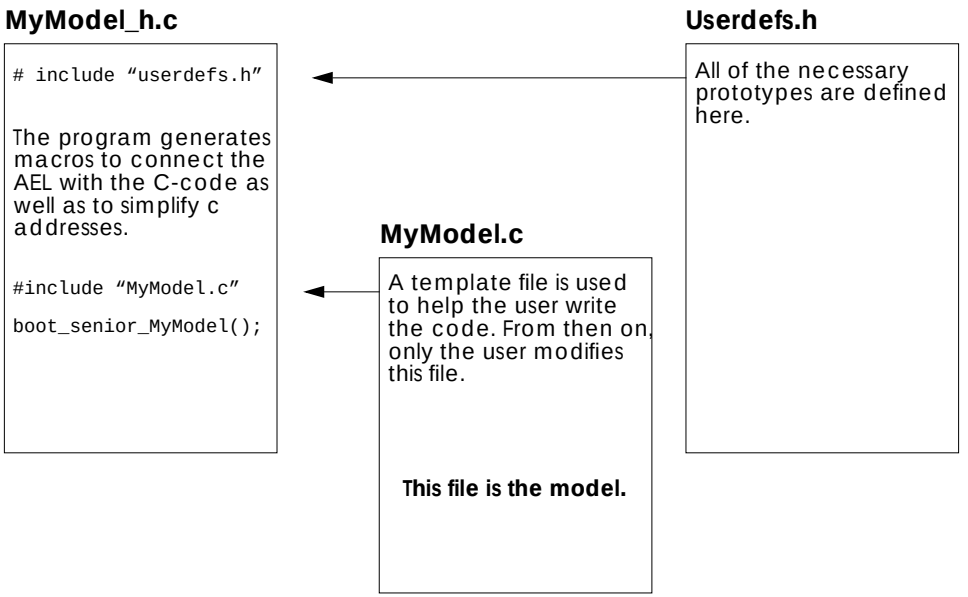


Figure 1-2. C-code File Relationship

The Makefile compiles the *MyModel_h.c* file which is autogenerated when the *Compile* button is clicked and the file is out-of-date relative to dialog box settings. It includes the header files and the user's code.

During the compile process the Makefile sources the *hpeesofdebug.mak* (or *hpeesofopt.mak*) file. This file includes the autogenerated modelbuilder.mak file and the user-editable user.mak file. The Model Builder interface executes the make process; the user can manually build the program with the following command:

hpeesofmake -f hpeesofdebug.mak <*target*>

where <*target*> is one of the following:

compile_only	compile only the active model.
link_only	link without compiling anything.
compile_and_link	compile the active file and link everything.
update_only	compile the active and any out-of-date files (including those in *user.mak*).
update_and_link	compile the active and any out-of-date files (including those in *user.mak*) and then link.

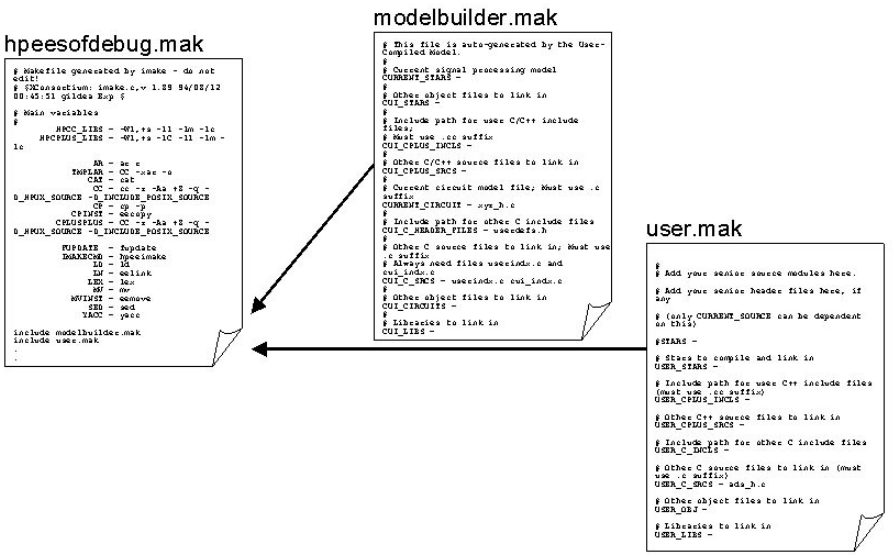


Figure 1-3. Makefile Relationship.

Accessing Dynamically Loaded Devices

When you create a dynamically-loaded device from ADS, everything is handled automatically. However a dynamically-loaded device, by default, is only accessible within the project in which it is created. The procedure that follows describes how to make a device available to other projects or users.

1. The simulator has a default list of directories to search, when looking for dynamically-loaded devices, as follows:

   ../networks

   $HOME/hpeesof/circuit/lib.$ARCH

   $HPEESOF_DIR/custom/circuit/lib.$ARCH

   $HPEESOF_DIR/circuit/lib.$ARCH

   These directories are searched in the order listed. Change the default path by setting the variable EESOF_MODEL_PATH in either of the following ways:

   $HPEESOF_DIR/custom/config/hpeesofsim.cfg

   $HOME/hpeesof/config/hpeesofsim.cfg

   For example (the default setting), see the entry in:

   $HPEESOF_DIR/config/hpeesofsim.cfg

2. Copy the dynamically-loaded device to one of the directories listed in EESOF_MODEL_PATH (see step 1).

3. In the directory where the dynamically-loaded device was copied, the following command must be executed:

   hpeesofsim -X

This will start the simulator, but instead of running a simulation, the current directory will be scanned for dynamically-loaded devices, and an index file (*deviceidx.db)* will be created. Copying a dynamically-loaded device to a directory is not enough. The directory dynamically-loaded device index must also be updated to include the new device. If this is not done, the simulator will be unable to locate the dynamically-loaded device. No simulator licenses of any type are required for this.

Note To run *hpeesofsim*, you must have*$HPEESOF_DIR/bin* in $PATH, and you must also have set the appropriate environment variable to tell your system about the ADS shared libraries/DLLs.

Chapter 2: Creating Linear Circuit Elements

This chapter describes creating linear elements through the use of examples. A linear element differs from a nonlinear element in that a linear element contains only linear elements while a nonlinear element can contain both linear and nonlinear elements.

Deriving S-Parameter Equations

One way to characterize a circuit element is by its S-parameters. To help you derive the S-parameters, refer to the following book: *Microwave Transistor Amplifiers* by Guillermo Gonzalez (Englewood Cliffs: Prentice-Hall, Inc., 1984).

Begin the process of deriving the S-parameters by examining the circuit configurations shown in Figure 2-1. Although this example shows 2-port S-parameters, the technique is the same for elements with a greater number of ports.

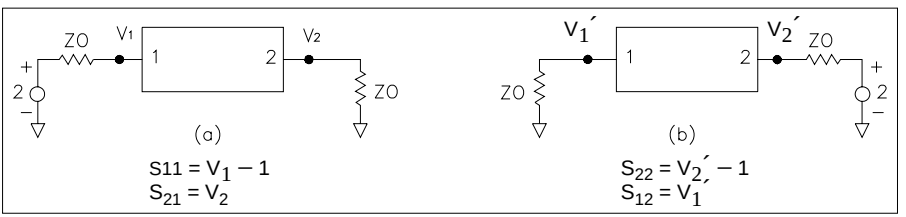


Figure 2-1. 2-port network

Alternative, but equivalent, expressions for S_{11} and S_{22} are

S_{11} = (Z1 – ZO)/(Z1 + ZO)

S_{22} = (Z2 – ZO)/(Z2 + ZO)

where

ZO is the normalizing impedance for the circuit (usually 50 ohms)

Z1 is the impedance looking into port 1 when port 2 is terminated with ZO

Z2 is the impedance looking into port 2 when port 1 is terminated with ZO

For example, consider a grounded pi-section resistive attenuator as shown in Figure 2-2. Inserting Figure 2-2 into the two-port network in Figure 2-1 results in Figure 2-3.

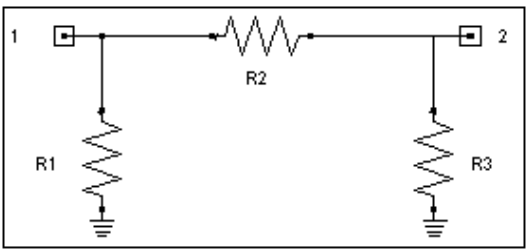


Figure 2-2. Schematic for pi-section resistive attenuator

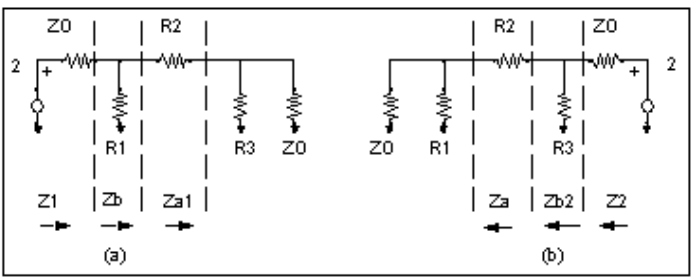


Figure 2-3. Resulting circuit schematic

Using Figure 2-3, the following relations are defined:

YA1 = 1.0/R3 + 1.0/ZO

ZA1 = 1.0/YA1

ZB1 = R2 + ZA1

Because the network is symmetrical, the following relations also hold:

YA2 = 1.0/R1 + 1.0/ZO

ZA2 = 1.0/YA2

ZB2 = R2 + ZA2

From the definition of Z1 and Z2:

Z1 = (R1• ZB1)/(R1 + ZB1)

Z2 = (R3 • ZB2)/(R3 + ZB2)

The S-parameters are obtained from the following equations:

S_{11} = (Z1 – ZO)/(Z1 + ZO)

S_{22} = (Z2 – ZO)/(Z2 + ZO)

$S_{12} = S_{21}$ = (2.0 / ZO)/(1.0 / ZA1+1.0 / ZA2 + R2 / (ZA1• ZA2))

These basic equations are sufficient to write the C function for the element.

Deriving Y-Parameter Equations

Y-parameters equations can be used to describe a user-defined element as an alternative to S-parameter equations. Figure 2-4 shows Y-parameters for a resistor connected between two ports; Y-parameter definitions follow the figure.

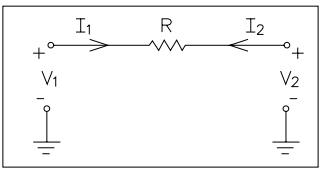


Figure 2-4. Y-parameters for a 2-port resistor connection

In general,

$$Y_{11} = \left.\frac{I_1}{V_1}\right|_{V_2 = 0} \qquad Y_{21} = \left.\frac{I_2}{V_1}\right|_{V_2 = 0}$$

$$Y_{12} = \left.\frac{I_1}{V_2}\right|_{V_1 = 0} \qquad Y_{22} = \left.\frac{I_2}{V_2}\right|_{V_1 = 0}$$

With V_2 equal to zero, $V_1 = I_1R$, which is also equal to $-I_2R$. Y_{11} reduces to 1/R and Y_{21} to –1/R. Setting V_1 to zero, $V_2 = I_2R = -I_1R$. The expressions for Y_{22} and Y_{12} are 1/R and –1/R, respectively. The resultant Y-parameter matrix is:

$$[Y] = \begin{bmatrix} 1/R & -1/R \\ -1/R & 1/R \end{bmatrix}$$

The following code is a portion of the example file:

```
/*************************************************************************
/
# define EPS 1.0e-8
/*
 * This example shows direct Y-parameter loading, instead of S-parameters.
 * For some elements, admittance parameters are easier to derive than
 * scattering parameters. For a series resistor, the admittance matrix is
 * as follows:
 *                         | g -g|
 *                   |Y| = |-g  g|   where g = 1 / R
 *
 * ELEMENT U2PD Id n1 n2 R=#
 */
static boolean u2pd_y(
UserInstDef *userInst,
double omega,
COMPLEX *yPar)
{
   double res, cond;
    res = userInst->pData->value.dVal * get_runit(userInst->eeElemInst);
    if (res < EPS)
       res = EPS;
    cond = 1.0 / res;
    yPar[0].real = yPar[3].real = cond;
    yPar[1].real = yPar[2].real = -cond;
    yPar[0].imag = yPar[1].imag = yPar[2].imag = yPar[3].imag = 0.0;
    return TRUE;

}
#undef EPS
/*************************************************************************
/
```

Coding a Linear Element

Your circuit simulator includes examples of linear user-compiled models. You can follow the same style in your modules. You can define only one model per module. Every model includes a *_h.c file, which contains macros, type definitions, and interface function declarations. If you are interested you can study this file to learn how dialog box settings map to the c-code. Note that the file is automatically generated so any changes made directly to the file will be lost.

To create a linear element, perform the following steps:

1. Define the element from the parameters page and define the number of external pins from the Symbol View (accessed from the Model Code tab).
2. Write the function to return the linear response. The linear behavior is characterized by a linear analysis function that you will write; this corresponds to the `compute_y` function pointer in the `UserElemDef` structure (already defined in the template code file):

   ```
   boolean (*compute_y)(UserInstDef *pInst, double omega, COMPLEX *yPar)
   ```

   This function must return `TRUE` if successful, `FALSE` otherwise. This function should be capable of working at $\omega = 0$, especially if it is used for convolution. You can use Y-parameters directly, or compute S-parameters and call the supplied `s_y_convert` function to obtain Y-parameters:

   ```
   extern boolean s_y_convert(COMPLEX *inPar, COMPLEX *outPar, int
   direction, double rNorm, int size)
   ```

3. Write the function to return the linear noise response. The linear noise behavior is characterized by a noise analysis function; this corresponds to the compute_n function pointer (already defined in the template code file):

   ```
   boolean (*compute_n)(UserInstDef *pInst, double omega, COMPLEX *yPar,
   COMPLEX *nCor);
   ```

   It must compute the N×N COMPLEX noise correlation matrix using the passed arguments omega and yPar array. The Code Options dialog box setting will set this to NULL if the element is noiseless. The function must return TRUE if successful, FALSE otherwise.

   Thermal noise generated by a user-defined passive n-port element (where n is between 1 and 20) at some element temperature tempC deg. Celsius can be included in the nodal noise analysis of the parent network by calling the provided function:

   ```
   boolean passive_noise(COMPLEX *yPar, double tempC, int numNodes, COMPLEX
   *nCor)
   ```

   from the element's compute_n function.

   For an active 3-terminal 2-port element, if the conventional 2-port noise parameters (minimum noise figure, optimum source reflection coefficient, effective noise resistance) are available through a measured data file, the 2×2 COMPLEX noise correlation matrix required by compute_n can be obtained using the provided function:

   ```
   boolean active_noise (COMPLEX *yPar, NParType *nPar, int numFloatPins,
   COMPLEX *nCor)
   ```

   numFloatPins is either 3 for floating reference pin, or 2 for grounded reference pin. You must fill the noise parameters into the nPar structure.

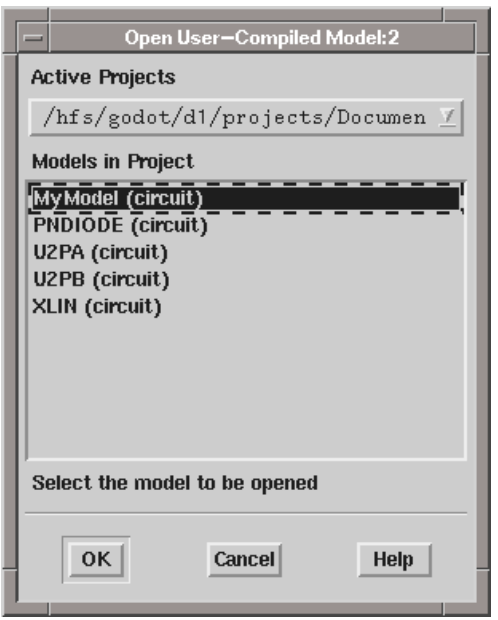
4. If the element needs special pre-analysis processing, such as reading data/technology files, the `pre_analysis` pointer must be set to an appropriate processing function. The Code Options dialog box value will determine whether this pointer is set to `NULL` or to the `pre_analysis` function. The function must return `TRUE` if successful, `FALSE` otherwise.
5. Before the beginning of a new circuit analysis, you must write the function for any cleanup or post-processing required by the element (such as freeing memory or writing an output file) and set the Post-Analysis Function check box in the Code Options dialog. The function must return `TRUE` if successful, `FALSE` otherwise.
6. To allow detailed or extra information in your user-defined element definition, the pointer field `seniorData` can be used to point to an arbitrary structure.

Pi-Section Resistive Attenuator

The steps in the preceding section “Coding a Linear Element” on page 2-6 are described for the grounded pi-section resistive attenuator example U2PA in the following sections.

Element Definition

The array *U2PA* (with static or module scope) defines the parameters of the U2PA element. The c code is automatically generated by the information in the dialog box.



The corresponding c-headers are automatically generated:

```
#define R1_P  userInst->pData[0].value.dVal
#define R2_P  userInst->pData[1].value.dVal
#define R3_P  userInst->pData[2].value.dVal
static UserParamType
U2PA_parms[] =
{
  {“R1”, REAL_data},  {“R2”, REAL_data},  {“R3”, REAL_data}
};
static UserElemDef U2PA_ELEMENTS[] =
{
    “U2PA”,   /* modelName */
    NUM_EXT_NODES,          /* # of external nodes */
    siz(U2PA_PARMS),   /* # of parameters */
    U2PA_PARMS,   /* # of parameter structure */
    PRE_ANALYSIS_FCN_PTR,   /* pre-analysis fcn ptr */
    COMPUTE_Y_FCN_PTR,      /* Linear model fcn ptr */
    COMPUTE_N_FCN_PTR,      /* Linear noise model fcn ptr */
    POST_ANALYSIS_FCN_PTR,  /* post-analysis fcn ptr */
    NULL,                /* nonlinear structure ptr */
    NULL,                   /* User-defined arb. data structure */
};
```

It is up to the user to write the appropriate code for the `compute_y` and `compute_n` functions.

Defining Variables

Begin by defining the variables and their data types. The S-parameter equations derived in Figure 2-3 provide the basis for the needed variables. The equations are repeated here:

YA1 = 1.0/R3 + 1.0/ZO

ZA1 = 1.0/YA1

ZB1 = R2 + ZA1

YA2 = 1.0/R1 + 1.0/ZO

ZA2 = 1.0/YA2

ZB2 = R2 + ZA2

Z1 = (R1 × ZB1)/(R1 + ZB1)

Z2 = (R3 × ZB2)/(R3 + ZB2)

S_{11} = (Z1 - ZO)/(Z1 + ZO)

S_{22} = (Z2 - ZO)/(Z2 + ZO)

$S_{12} = S_{21}$ = (2.0 / ZO)/(1.0 / ZA1+1.0 / ZA2 + R2/(ZA1 • ZA2))

The resulting declarations are:

```
double YA1, YA2;
double ZA1, ZA2, ZB1, ZB2;
double Z1, Z2;
COMPLEX S[4];
```

Implementing S-Parameter Equations

Implement the equations by performing the following steps:

1. The parameters are available via macro definitions as the parameter name, with an appended _P:

```
R1 = R1_P;
R2 = R2_P;
R3 = R3_P;
```

2. Include code to check the resistance values and protect against division by zero.

   In this example, the expressions YA1 and YA2 demonstrate the need to check data values. If R1 or R3 has a value of zero, a fatal division by zero error condition will result.

   To protect against division by zero, limit the lower value of all input parameters to an arbitrarily low value. For easy use, assign the value to a C macro, for example, EPS to mean 10^{-8}.

```
#define EPS 1.0E-8
if (R1 < EPS)
  R1 = EPS;
if (R2 < EPS)
  R2 = EPS;
if (R3 < EPS)
  R3 = EPS;
```

3. Insert code to define the equations. Note that the 2×2 Y-parameters to be returned in the `yPar[0..3]` locations must be in row order, for example, Y_{11}, Y_{12}, Y_{21} and Y_{22}.

```
S[3].imag = S[2].imag = S[1].imag = S[0].imag = 0.0; /* imag part */
YA1 = 1.0 / R3 + 1.0 / ZO;
ZA1 = 1.0 / YA1;
ZB1 = R2 + ZA1;
Z1 = (R1 * ZB1) / (R1 + ZB1);
S[0].real = (Z1 - ZO) / (Z1 + ZO); /* S11 real */

YA2 = 1.0 / R1 + 1.0 / ZO;
ZA2 = 1.0 / YA2;
ZB2 = R2 + ZA2;
Z2 = (R3 * ZB2) / (R3 + ZB2);
S[3].real = (Z2 - ZO) / (Z2 + ZO); /* S22 */
S[2].real = S[1].real = (2.0/ZO) /
(1.0/ZA1 + 1.0/ZA2 + R2/(ZA1 * ZA2));

/* convert S[2x2] -> yPar[2x2] */
return s_y_convert(S, yPar, 1, ZO, 2);

return status;
```

Adding Noise Characteristics

The noise analysis function pointer compute_n for passive elements in this example is set to thermal_n, which computes thermal noise of the element item at the simulator default temperature of 27.0°C, as shown below.

```
#define STDTEMP 27.0
/*
 * Thermal noise model at default temperature (27.0 deg.C) for any
 * n-terminal linear element
 */
static boolean thermal_n(
UserInstDef *userInst,
  double omega,
  COMPLEX *yPar,
  COMPLEX *nCorr)
{
  UserElemDef *userDef = userInst->userDef;
return passive_noise(yPar, STDTEMP, userDef->numExtNodes, nCorr);
}
```

The passive_noise function uses the supplied N×N Y-parameters of an N-terminal element and temperature to compute the N×N complex noise correlation matrix.

It is possible to compute thermal noise at variable temperatures by adding a temperature parameter, which could be either REAL_data or a MTRL_data reference, to the element definition.

The next step is compiling and linking the C code. Refer to Chapter 1, Building User-Compiled Analog Models.

Transmission Line Section

This example will show how to derive an S-parameter matrix for a general transmission line section, then show how to apply this to the case of a coaxial cable. The end result will be an element that can produce an S-parameter matrix given physical dimensions.

Deriving an S-Parameter

The ABCD matrix for a general section of lossless transmission line is:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} \cos\left(\frac{2\pi L}{\lambda}\right) & jZ\sin\left(\frac{2\pi L}{\lambda}\right) \\ \frac{j}{Z}\sin\left(\frac{2\pi L}{\lambda}\right) & \cos\left(\frac{2\pi L}{\lambda}\right) \end{bmatrix}$$

From *Microwave Transistor Amplifiers* by G. Gonzalez, the conversion from an ABCD to an S-matrix is as follows:

$$\begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix} = \begin{bmatrix} \frac{A' + B' - C' - D'}{\Delta} & \frac{2 \bullet (A'D' - B'C')}{\Delta} \\ \frac{2}{\Delta} & \frac{-A' + B' - C' + D'}{\Delta} \end{bmatrix}$$

where:

$$A' = A = \cos\left(\frac{2\pi L}{\lambda}\right) \qquad C' = CZ_o = j\left(\frac{Z}{Z_o}\right)\sin\left(\frac{2\pi L}{\lambda}\right)$$

$$B' = \frac{B}{Z_o} = j\left(\frac{Z}{Z_o}\right)\sin\left(\frac{2\pi L}{\lambda}\right) \qquad D' = D = \cos\left(\frac{2\pi L}{\lambda}\right)$$

and

$$\Delta = A' + B' + C' + D'.$$

Define β as $\frac{2\pi}{\lambda}$. Then $\beta L = \frac{2\pi L}{\lambda}$.

When the mathematical equations are worked out, this leaves:

$$S_{11} = S_{22} = \frac{j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) - \left(\frac{Z_o}{Z}\right)\right]}{2\cos(\beta L) + j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]}$$

and

$$S_{12} = S_{21} = \frac{2}{2\cos(\beta L) + j\sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]}$$

Separating the Expressions

Because we need S-parameters in a real-imaginary format, we need to separate these expressions into their real and imaginary parts by applying their complex conjugate:

$$S_{11} = S_{22} = \frac{j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) - \left(\frac{Z_o}{Z}\right)\right]}{2\cos(\beta L) + j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]} \bullet \frac{2\cos(\beta L) - j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]}{2\cos(\beta L) - j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]}$$

This multiplication yields:

$$Re[S_{11}] = \frac{\sin^2(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right)^2 - \left(\frac{Z_o}{Z}\right)^2\right]}{4\cos^2(\beta L) + \sin^2(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]^2} = Re[S_{22}]$$

$$Im[S_{11}] = \frac{2\sin(\beta L)\cos(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) - \left(\frac{Z_o}{Z}\right)\right]}{4\cos^2(\beta L) + \sin^2(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]^2} = Im[S_{22}]$$

and

$$S_{12} = S_{21} = \frac{2}{2\cos(\beta L) + j\sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]} \bullet \frac{2\cos(\beta L) - j\sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]}{2\cos(\beta L) - j\sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]}$$

This multiplication yields:

$$Re[S_{21}] = \frac{4\cos(\beta L)}{4\cos^2(\beta L) + \sin^2(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]^2} = Re[S_{12}]$$

$$Im[S_{21}] = \frac{-2\sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) - \left(\frac{Z_o}{Z}\right)\right]}{4\cos^2(\beta L) + \sin^2(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right)\right]^2} = Im[S_{12}]$$

Note that the expressions all have the same denominator, which makes them easier to code.

Algorithms

Let us go through an algorithm with an example to ensure that it is correct.

Take Z = 50 Ω, Z_o = 50 Ω, and L = λ. Then βL = 2 π. This yields:

$$Re[S_{11}] = 0. \quad (\sin(2\pi) = 0)$$

$$Im[S_{11}] = \frac{2\sin(2\pi)\cos(2\pi) \bullet \left[\left(\frac{50}{50}\right) - \left(\frac{50}{50}\right)\right]}{4\cos^2(2\pi) + \sin^2(2\pi) \bullet (1+1)^2} = 0$$

$$Re[S_{21}] = \frac{4\cos(2\pi)}{4\cos^2(2\pi) + \sin^2(2\pi) \bullet (1+1)^2} = 1$$

$$Im[S_{21}] = \frac{-2\sin(2\pi) \bullet (1+1)}{4\cos^2(2\pi) + \sin^2(2\pi) \bullet (1+1)^2} = 0$$

Another example: Z = Zo = 50 Ω, β L = π /2 (90°)

$$Re[S_{11}] = \frac{\sin^2\left(\frac{\pi}{2}\right) \bullet (1^2 - 1^2)}{4\cos^2\left(\frac{\pi}{2}\right) + \sin^2\left(\frac{\pi}{2}\right) \bullet (1+1)^2} = 0$$

$$Im[S_{11}] = \frac{2\sin\left(\frac{\pi}{2}\right)\cos\left(\frac{\pi}{2}\right) \bullet (1-1)}{4\cos^2\left(\frac{\pi}{2}\right) + \sin^2\left(\frac{\pi}{2}\right) \bullet (1+1)^2} = 0$$

$$Re[S_{21}] = \frac{4\cos\left(\frac{\pi}{2}\right)}{4\cos^2\left(\frac{\pi}{2}\right) + \sin^2\left(\frac{\pi}{2}\right) \bullet (1+1)^2} = 0$$

$$Im[S_{21}] = \frac{-2\left(\frac{\pi}{2}\right)(1+1)}{4\cos^2\left(\frac{\pi}{2}\right) + \sin^2\left(\frac{\pi}{2}\right) \bullet (1+1)^2} = -1$$

The preceding expressions can be used with any transmission line section as long as b (the propagation constant), Z (the impedance), and L (the length) are known.

Applying a Problem to the Coaxial Cable Section

The impedance of a coaxial cable is defined by:

$$Z = \frac{\eta}{2\pi} ln\left(\frac{B}{A}\right)$$

where:

Z = impedance

η = characteristic impedance of dielectric = $\sqrt{\frac{\mu_o}{\varepsilon}}$

B = outside diameter

A = inside diameter

The propagation constant is defined by:

$$\beta = \frac{2\pi}{\lambda} \quad where\ \lambda = \frac{speed\ \text{of light in vacuum}}{\sqrt{E_r} \bullet FREQ}$$

Therefore, the required parameters are A, B, L, and E_r.

The U2PB example in U2PB.c is an implementation of the above. The relevant defining data structures are shown below:

```
static UserParamType
U2PB_parms[] =
{
    {“A”, REAL_data},  {“B”, REAL_data},  {“L”, REAL_data},
    {“K”, REAL_data}
};

static UserElemDef U2PB_ELEMENTS[] =
{
    “U2PB”,   /* modelName */
    NUM_EXT_NODES,          /* # of external nodes */
    siz(U2PB_PARMS),   /* # of parameters */
    U2PB_PARMS,   /* # of parameter structure */
    PRE_ANALYSIS_FCN_PTR,   /* pre-analysis fcn ptr */
    COMPUTE_Y_FCN_PTR,      /* Linear model fcn ptr */
    COMPUTE_N_FCN_PTR,      /* Linear noise model fcn ptr */
    POST_ANALYSIS_FCN_PTR,  /* post-analysis fcn ptr */
    NULL,                /* nonlinear structure ptr */
    NULL,  /* User-defined arb. data structure */
};
```

The a, b, len, and k values are obtained from the circuit through the automatically defined macros:

a = A_P;

b = B_P;

len = L_P;

k = K_P;

To prevent the program from crashing, some error trapping must be done: *a* is checked to be positive; b is checked to be greater than a; and, k is checked to be greater than or equal to *1*.

```
if (a <= 0.0 || b <= a || k < 1)
{
  (void)sprintf(ErrMsg, "u2pb_y(%s): invalid params: A=%g,
  B=%g,K=%g",userInst->tag, a, b, k);
  send_error_to_scn(ErrMsg);
  return FALSE;
}
```

Calculating Remaining Expressions

The impedance and wave number are then calculated:

```
eta = sqrt(MU0/EPS0/k);
vphase = 1.0 / sqrt(MU0 * EPS0 * k);
betal = omega * len / vphase;
z = eta * log(b / a) / 2.0 / PI;
```

The remaining expressions calculate the S-matrix:

```
zzo=  z / ZO;
zoz=  ZO / z;
arg1=  zzo - zoz;
arg2=  zzo + zoz;
denom = 4.0 * sqr (cos(betal)) + sqr (sin(betal)) * sqr (arg2);
res11 = sqr (sin(betal)) * (sqr(zzo) -sqr (zoz)) / denom;
ims11 = 2.0 * sin(betal) * cos (betal) * arg1 / denom;
res21 = 4.0 * cos(betal) / denom;
ims21 = -2.0 * sin(betal) * arg2 / denom;
S[3].real = S[0].real = res11; (defines S11 .real, S22 .real)
S[3].imag = S[0].imag = ims11; (defines S11 .imag, S22 .imag)
S[2].real = S[1].real = res21; (defines S12 .real, S21 .real)
S[2].imag = S[1].imag = ims21; (defines S12 .imag, S21 .imag)
```

Adding Noise Characteristics

Because the U2PB coaxial section is lossless, it is also noiseless; therefore, the *Noise Analysis Function* check box in the Code Options dialog box is not selected.

The next step is compiling and linking the C-code. See Chapter 1, Building User-Compiled Analog Models.

Chapter 3: Creating Nonlinear Circuit Elements

This chapter describes creating nonlinear circuit elements. Nonlinear user-defined element modeling described in this chapter is applicable to steady-state analysis only. Refer to Chapter 4, Creating Transient Circuit Elements, for information on modeling the transient response.

Requirements for Creating Nonlinear Elements

A nonlinear circuit element is characterized as follows:

- a linear part
- a nonlinear part
- a bias-dependent small-signal ac part
- a bias-dependent noise part

The first two are mutually exclusive partitions of the element model, while the small-signal part is a combination. All parts can use parameter data of the element item as well as those of any referenced items in the circuit.

The parts are coded as the following function entries in the element's device definition:

- *analyze_lin*
- *analyze_nl*
- *analyze_ac*
- *analyze_ac_n*

Linear Part

The linear part is computed in frequency domain. The user code must compute the branch admittances in the same way as in the linear element case (Refer to Chapter 2, Creating Linear Circuit Elements). The difference here is in the loading of the circuit nodal admittance matrix, which must be done through the `add_lin_y` function call for each contribution separately. The `analyze_lin` function is called once for every sweep (frequency, power, swept variable) value. The function can be set to `NULL` if the element is completely nonlinear.

Nonlinear Part

The nonlinear part is evaluated on a sample-by-sample basis of time domain pin voltages. The device's `analyze_nl` function must compute the instantaneous nonlinear currents, charges, and their voltage derivatives. Given the user item pin voltages—in the order of external followed by internal—the user-written code computes the nonlinear charges at each pin and the nonlinear currents out of each pin.

The partial derivatives of these nonlinear quantities with respect to each pin voltage are then computed to formulate conductances and capacitances to load into the circuit Jacobian matrix. In nonlinear analyses, the derivatives influence the rate of convergence, but have no effect on the final steady-state solution. In addition to the instantaneous voltages, delayed pin voltages can be obtained through the `get_delay_v` function.

You may keep any static/intermediate computed data (data invariant over subsequent time samples and iterations) with the particular element item itself. Functions within ADS perform the required time-to-frequency transformations.

AC Part

The ac part linearizes the element model around the dc bias point, and returns the small-signal frequency domain admittance and normalized noise correlation parameters.

The dc bias is determined for the entire flattened circuit, including nonlinear user element items, whose linear and nonlinear parts would be computed as above. Then the device's `analyze_ac` function is called to load the device admittances for all its branches (including internal) into the circuit nodal admittance matrix. If the conductances and capacitances are frequency independent, this will be a combination of the `analyze_lin` and linearized `analyze_nl` functions.

The `analyze_ac_n` function must load the bias-dependent noise current correlation parameters (normalized to `FOUR_K_TO`) using the interface function `add_lin_n`. It is called only if a noise measurement is required in a test bench.

The `UserElemDef` declaration for a nonlinear element has the `compute_y` function automatically set to `NULL`.

User-defined P-N Diode Model

This example shows how to create a nonlinear model of a P-N diode. The result is a set of functions (available in the example `PNDIODE` that provide a simplified model of the ADS `DIODE` element.

The model shown in Figure 3-1 is used for the PNDIODE element.

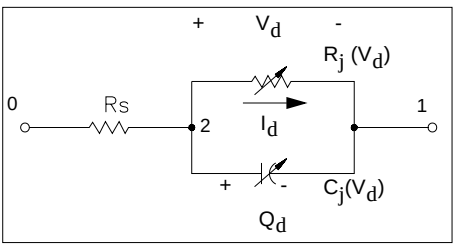
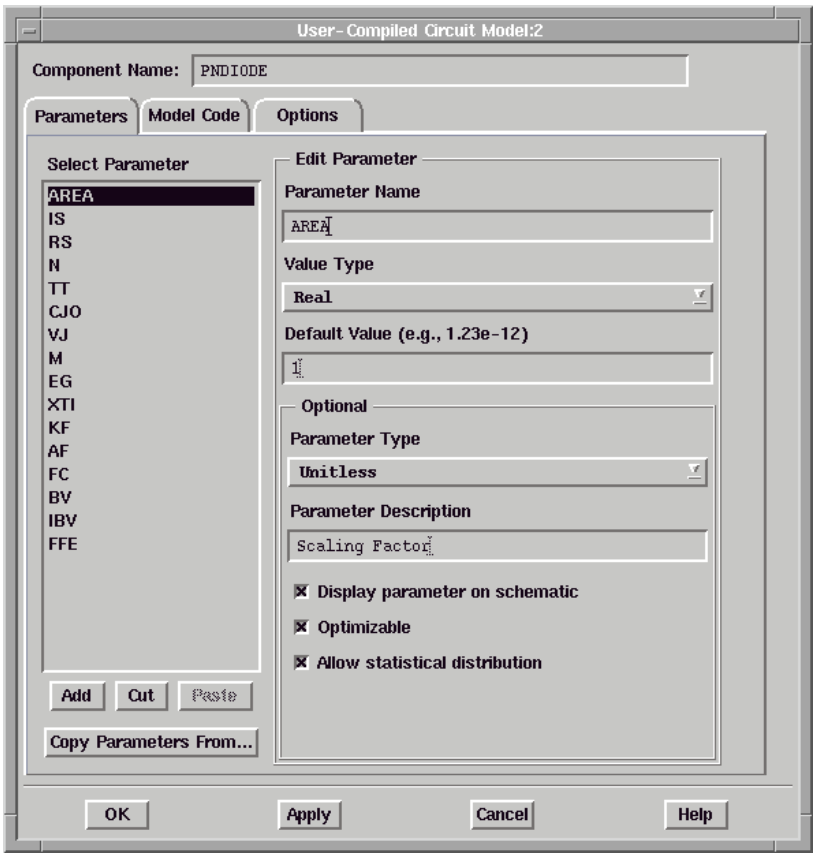


Figure 3-1. PNDIODE element model

Defining a Nonlinear Element

For simplicity, all diode parameters are defined with the element itself in the `UserParamType` array `PNDIODE`, instead of an indirect, shareable model form reference (in Series IV these were referred to as data items). These definitions are entered in the Parameters tab dialog box:



The associated declarations are automatically generated in the PNDIODE_h.c file:

```
static UserParamType
PNDIODE_parms[] =
{
    {“AREA”, REAL_data},  {“IS”, REAL_data},  {“RS”, REAL_data},
    {“N”, REAL_data},  {“TT”, REAL_data},
    {“CJO”, REAL_data},  {“VJ”, REAL_data},
    {“M”, REAL_data},  {“EG”, REAL_data},
    {“XTI”, REAL_data},  {“KF”, REAL_data},
    {“AF”, REAL_data},  {“FC”, REAL_data},
    {“BV”, REAL_data},  {“IBV”, REAL_data},
    {“FFE”, REAL_data}
};
```

The associated AEL create_item declarations are also generated in the PNDIODE.ael file.

The three function entries required in a user nonlinear device definition are declared automatically:

```
static boolean analyze_lin(UserInstDef *userInst, double omega);

static boolean analyze_nl(UserInstDef *userInst, double *vPin);

static boolean analyze_ac(UserInstDef *userInst, double *vPin, double
omega)
```

The diode has one internal pin between the linear RS and the nonlinear R || C representing the junction. The device definition is (again, automatically-generated):

```
#define ANALYZE_NL_FCN_PTR      analyze_nl
#define ANALYZE_LIN_FCN_PTR     analyze_lin
#define ANALYZE_AC_FCN_PTR      analyze_ac
#define NUM_NONLINEAR_INT_NODES 1
#define ANALYZE_AC_N_FCN_PTR    NULL
static UserNonLinDef
ANALYZE_NL_DEF_PTR =
{
    NUM_NONLINEAR_INT_NODES,   /* numIntNodes */
    ANALYZE_LIN_FCN_PTR,       /* analyze_lin() */
    ANALYZE_NL_FCN_PTR,        /* analyze_nl() */
   ANALYZE_AC_FCN_PTR,        /* analyze_ac() */
   NULL,                      /* Nonlin modelDef (user can change) */
    ANALYZE_AC_N_FCN_PTR,      /* analyze_ac_n() */
};
```

The entry for the diode element definition itself is completed, using its parameters and device definition:

```
#define NUM_EXT_NODES           2
#define ANALYZE_NL_DEF_PTR      analyze_nl_def_ptr
#define COMPUTE_Y_FCN_PTR       NULL
#define PRE_ANALYSIS_FCN_PTR    NULL
#define POST_ANALYSIS_FCN_PTR   NULL
#define ANALYZE_TR_FCN_PTR      NULL
#define ANALYZE_AC_N_FCN_PTR    NULL
#define COMPUTE_N_FCN_PTR       NULL
#define PNDIODE_PARMS                 PNDIODE_parms
#define PNDIODE_ELEMENTS              PNDIODE_elements
static UserElemDef PNDIODE_ELEMENTS[] =
{
   “PNDIODE”,   /* modelName */
   NUM_EXT_NODES,          /* # of external nodes */
    siz(PNDIODE_PARMS),   /* # of parameters */
    PNDIODE_PARMS,   /* # of parameter structure */
    PRE_ANALYSIS_FCN_PTR,   /* pre-analysis fcn ptr */
    COMPUTE_Y_FCN_PTR,      /* Linear model fcn ptr */
    COMPUTE_N_FCN_PTR,      /* Linear noise model fcn ptr */
    POST_ANALYSIS_FCN_PTR,  /* post-analysis fcn ptr */
    &ANALYZE_NL_DEF_PTR,    /* nonlinear structure ptr */
    NULL,                   /* User-defined arb. data structure */
};
```

Implementation of the preceding functions is described next. (Error message reporting, while useful for debugging, is not shown below.) The C macros conveniently centralize parameter indexing:

```
#define AREA_P  userInst->pData[0].value.dVal
#define IS_P  userInst->pData[1].value.dVal
#define RS_P  userInst->pData[2].value.dVal
#define N_P  userInst->pData[3].value.dVal
#define TT_P  userInst->pData[4].value.dVal
#define CJO_P  userInst->pData[5].value.dVal
#define VJ_P  userInst->pData[6].value.dVal
#define M_P  userInst->pData[7].value.dVal
#define EG_P  userInst->pData[8].value.dVal
#define XTI_P  userInst->pData[9].value.dVal
#define KF_P  userInst->pData[10].value.dVal
#define AF_P  userInst->pData[11].value.dVal
#define FC_P  userInst->pData[12].value.dVal
#define BV_P  userInst->pData[13].value.dVal
#define IBV_P  userInst->pData[14].value.dVal
#define FFE_P  userInst->pData[15].value.dVal
```

The linear contribution is just from the series resistor RS between pins 0 and 1. This is coded in the *analyze_lin* function:

```
static boolean analyze_lin (
   UserInstDef *userInst,
   double omega)
{
   boolean status;
   COMPLEX y;
   UserParamData *pData = userInst->pData;

    y.real = y.imag = 0.0;
    if (RS_P > RMIN)
     y.real = AREA_P / RS_P;
    else
     y.real = GMAX;

    status = add_y_branch(userInst, 0, 2, y);

    if (!status)
    {
        (void)sprintf(ErrMsg, “analyze_lin(%s) -> add_lin_y() failed”,
                    userInst->tag);
      send_error_to_scn(ErrMsg);
    }
    return status;
}
```

This function is also called from the small-signal `analyze_ac` function described later.

The nonlinear device model is coded as a common function (diode_nl_iq_gc that follows) so that it can be called from both analyze_nl and analyze_ac. It computes the nonlinear junction charge, current and their derivatives with respect to the junction voltage.

```
static void diode_nl_iq_gc (
  UserInstDef *userInst, /* Changed from SIV to be consistent w/CUI */
  double *vPin,
  double *id,
  double *qd,
  double *gd,
  double *capd)
{
    double vd, csat, vte, evd, evrev;
    double exparg;
    double fcpb, xfc, f1, f2, f3;
    double czero, arg, sarg, czof2;
    UserParamData *pData = userInst->pData;

    csat = IS_P * AREA_P;
    vte = N_P * VT;
    vd = vPin[2] - vPin[1]; /* junction voltage */

  /*
   * compute current and derivatives with respect to voltage
   */
    if ( vd >= -5.0*vte )
    {
       exparg = ( vd/vte < 40.0 ) ? vd/vte : 40.0;
       evd = exp(exparg);
      *id = csat * (evd - 1.0) + GMIN * vd;
      *gd = csat * evd / vte + GMIN;
    }
    else
    {
      *id = -csat + GMIN * vd;
      *gd = -csat / vd + GMIN;
      if ( BV_P != 0.0 && vd <= (-BV_P+50.0*VT) )
      {
           exparg = ( -(BV_P+vd)/VT < 40.0 ) ? -(BV_P+vd)/VT : 40.0;
        evrev = exp(exparg);
        *id -= csat * evrev;
        *gd += csat * evrev / VT;
      }
    }

  /*
```

```
   * charge storage elements
   */
  fcpb = FC_P * VJ_P;

czero = CJO_P * AREA_P;
  if (vd < fcpb)
  {
      arg = 1.0 - vd / VJ_P;
      sarg = exp(-M_P * log(arg));
        *qd = TT_P * (*id) + VJ_P * czero * (1.0 - arg * sarg)
         / (1.0 - M_P);
      *capd = TT_P * (*gd) + czero * sarg;
  }
  else
  {
      xfc = log(1.0 - FC_P);
    /* f1 = vj*(1.0-(1.0-fc)^(1.0-m))/(1.0-m)  */
      f1 = VJ_P * (1.0-exp((1.0-M_P)*xfc)) / (1.0-M_P);

    /* f2 = (1.0-fc)^(1.0+m)  */
      f2 = exp((1.0+M_P)*xfc);

    /* f3=1.0-fc*(1.0+m)  */
      f3 = 1.0 - FC_P * (1.0+M_P);
      czof2 = czero / f2;
      *qd = TT_P * (*id) + czero * f1 + czof2 * (f3 * (vd - fcpb) +
          (M_P / (VJ_P + VJ_P)) * (vd * vd - fcpb * fcpb));
      *capd = TT_P * (*gd) + czof2 * (f3 + M_P * vd / VJ_P);
  }
} /* diode_nl_iq_gc() */
```

The following equation is used for the diode current I_d in the forward bias mode:

$$I_d(V_d) = I_s\left(e^{\frac{V_d}{V_{te}}} - 1\right) + 10^{-12} \bullet V_d$$

The derivative of the diode current with respect to the junction voltage is:

$$\frac{dI_d}{dV_d} = \frac{Is \bullet e^{\frac{V_d}{V_{te}}}}{V_{te}} + 10^{-12}$$

For the case $V_d < FC \bullet VJ$, the total charge and large-signal incremental capacitance expressions are:

$$Q_d(V_d) = TT \bullet I_d + \frac{VJ \bullet CJO}{(1-M)} \bullet \left[1 - \left(1 - \frac{V_d}{VJ}\right)^{1-M}\right.$$

$$C_d(V_d) = \frac{dQ_d}{dV_d} = TT \bullet \frac{dI_d}{dV_d} + CJO\left(1 - \frac{V_d}{VJ}\right)^{-M}$$

The analyze_nl function that follows loads the nonlinear currents, charges at each pin, and the nonlinear conductances, capacitances for each branch. Note that each G, C component has four Jacobian matrix contributions, two diagonals and two off-diagonals.

```
static boolean analyze_nl (
  UserInstDef *userInst,
  double *vPin)
{
  double id, gd; /* current, conductance */
  double qd, capd; /* charge, capacitance */
  boolean status;
  char *pMsg = NULL;
  diode_nl_iq_gc(userInst, vPin, &id, &qd, &gd, &capd);
  /*
   * load nonlinear pin currents out of each terminal and
   * nonlinear charges at each terminal.
   */
  status = add_nl_iq(userInst, 1, -id, -qd) &&
           add_nl_iq(userInst, 2, id,  qd);
  if (!status)
  {
    pMsg = “add_nl_iq()”;
    goto END;
  }

  /* Add nonlinear conductance, capacitance
   *      0   1   2
   *   0
   *   1      Y   Y
   *   2      Y   Y
   */
  status = add_nl_gc(userInst, 1, 1,  gd, capd) &&
           add_nl_gc(userInst, 1, 2, -gd, -capd ) &&
           add_nl_gc(userInst, 2, 1, -gd, -capd ) &&
           add_nl_gc(userInst, 2, 2,  gd,  capd);

  if (!status)
    pMsg = “add_nl_gc()”;

 END:
  if (pMsg)
  {
    (void)sprintf(ErrMsg, “Error: PNDIODE: analyze_nl(%s) -> %s”,
userInst->tag, pMsg);
    send_error_to_scn(ErrMsg);
  }
```

```
  return status;
} /* analyze_nl() */
```

The analyze_ac function that follows characterizes the PNDIODE's bias-dependent small-signal ac behavior. It calls analyze_lin to load the linear part, then loads the linearized admittances obtained from the nonlinear junction conductance and capacitance at the DC bias point.

```
static boolean analyze_ac (
  UserInstDef *userInst,
  double *vPin,
  double omega)
{
  COMPLEX y;
  double id, gd; /* current, conductance */
  double qd, capd; /* charge, capacitance */
  boolean status;

  /*
   * Add linearized conductance, susceptance
   *      0   1   2
   *   0  G       G
   *   1      Y   Y
   *   2  G   Y   Y
   */
  if (!analyze_lin(userInst, omega))
    return pw;

  diode_nl_iq_gc(userInst, vPin, &id, &qd, &gd, &capd);
  y.real = gd; y.imag = omega * capd;
  status = add_y_branch(userInst, 1, 2, y);
  if (!status)
  {
    (void)sprintf(ErrMsg, “Error: PNDIODE: analyze_ac(%s) -> add_lin_y”,
userIns
t->tag);
    send_error_to_scn(ErrMsg);
  }
  return status;
} /* analyze_ac() */
```

The analyze_ac_n function that follows models the PNDIODE's noise behavior in a linear analysis. It loads the device's thermal noise, and its bias-dependent shot and flicker noise contributions through the interface primitive function add_lin_n. The static function add_n_branch loads a branch contribution symmetrically into the circuit's indefinite noise-current correlation matrix.

```
/*----------------------------------------------------------------------*/
static boolean add_n_branch(
  UserInstDef *userInst,
  int n1,
  int n2,
  COMPLEX iNcorr)
{
  boolean status = TRUE;

  status = add_lin_n(userInst, n1, n1, iNcorr) &&
           add_lin_n(userInst, n2, n2, iNcorr);
  if (status)
  {
    iNcorr.real = -iNcorr.real; iNcorr.imag = -iNcorr.imag;
    status = add_lin_n(userInst, n1, n2, iNcorr) &&
             add_lin_n(userInst, n2, n1, iNcorr);
  }
  return status;
}

/*----------------------------------------------------------------------*/
static boolean analyze_ac_n (
  UserInstDef *userInst,
  double *vPin,
  double omega)
{
  double id, gd; /* current, conductance */
  double qd, capd; /* charge, capacitance */
  boolean status;
  COMPLEX thermal, dNoise; /* noise-current correlation admittance */
  double kf, gs, tempScale;
  char *pMsg = NULL;
  UserParamData *pData = userInst->pData;

  diode_nl_iq_gc(userInst, vPin, &id, &qd, &gd, &capd);

  tempScale = DEV_TEMP / NOISE_REF_TEMP;

  dNoise.imag = thermal.imag = 0.0;

  if (RS_P > RMIN)
```

```
    gs = AREA_P / RS_P;
  else
    gs = GMAX;
  thermal.real = tempScale * gs;

  id = fabs(id);
  kf = fabs(KF_P);
  /* shot noise */
  dNoise.real = 2.0 * CHARGE * id;
  /* flicker noise */
  if (id > 0.0 && omega > 0.0 && kf > 0.0)
    dNoise.real += kf * pow(id, AF_P) * pow(omega/TWOPI, -FFE_P);

  dNoise.real /= FOUR_K_TO;

  status = add_n_branch(userInst, 0, 2, thermal) &&
           add_n_branch(userInst, 1, 2, dNoise);
  if (!status)
    pMsg = "add_lin_n()";

  if (pMsg)
  {
    (void)sprintf(ErrMsg, "Error: analyze_ac_n(%s) -> %s", userInst->tag,
pMsg);
    send_error_to_scn(ErrMsg);
  }
  return status;
} /* analyze_ac_n() */
```

The next step is compiling and linking the code. Refer to "Creating the Code and Compiling the Model" on page 1-9.

Referencing Data Items

Refer to "Referencing Data Items" on page 1-29.

Displaying Error/Warning Messages

Refer to "Displaying Error/Warning Messages" on page 1-30.

Chapter 4: Creating Transient Circuit Elements

This chapter describes the steps necessary for creating transient circuit elements.

Requirements for Creating Transient Elements

A transient model can be created for either a linear or nonlinear element. A transient element can have additional nodes internal to the element, as specified by the value of `numIntNodes` (set in the Code Options dialog field No. of internal nodes under the Transient Function check box. If a nonlinear model (using `UserNonLinDef`) is defined for the element, the `numIntNodes` in that structure must match this definition.

Special routines are available to simplify the use of ideal resistors, capacitors, inductors, and transmission lines within a transient element. For all but resistors, the number of these elements must be predefined using `numCaps`, `numInds`, and `numTlines` so the circuit simulator engine can perform the appropriate allocations. These values are entered in the appropriate fields in the Code Options dialog box.

The time-domain response is evaluated from the instantaneous pin voltages. The device's `analyze_tr` function must compute the instantaneous nonlinear currents, charges and their voltage derivatives. Given the instantaneous user item pin voltages (external first, followed by internal, starting at zero), the user-defined code computes the nonlinear charges at each pin and the nonlinear currents out of each pin. The partial derivatives of these nonlinear quantities with respect to each pin voltage are then computed to formulate conductances and capacitances to load into the circuit Jacobian matrix.

If P is the total number of pin voltages, formulate nonlinear current and charge at each pin n as follows:

$$r_n(t) = f(v_0(t), v_1(t), \ldots, v_{P-1}(t))$$

where r_n is the pin current or charge response.

These responses and their voltage derivatives (nonlinear conductances and capacitances) must be computed and loaded using the `add_tr_iq` and `add_tr_gc` functions, respectively.

Using Resistors, Capacitors, and Inductors

Special routines are available to simplify the use of ideal resistors, capacitors, inductors, and transmission lines within a transient element. These routines can be called from within the user-written `analyze_tr` function. For all but resistors, the number of these elements must be predefined using `numCaps`, `numInds` and `numTlines`, so the circuit simulator engine can perform the appropriate allocations.

If the requested number of elements is not used, an error message is generated and the analysis fails. The following code example could be used within an `analyze_tr` function to implement a transient model for the element model shown in Figure 4-1. Naturally, values for these components could be calculated from the `UserInstDef` parameters that are passed into the function.

```
boolean example1_tr (UserInstDef *pInst, double *vPin)
{
  boolean status;
  status = add_tr_resistor(pInst, 0, 2, 50.0) &&
       add_tr_resistor(pInst, 2, GND, 1000.0) &&
  add_tr_capacitor(pInst, 2, 1, 10.0e-12) &&
       add_tr_inductor(pInst, 2, 1, 5.0e-9);
  return status;
}
```

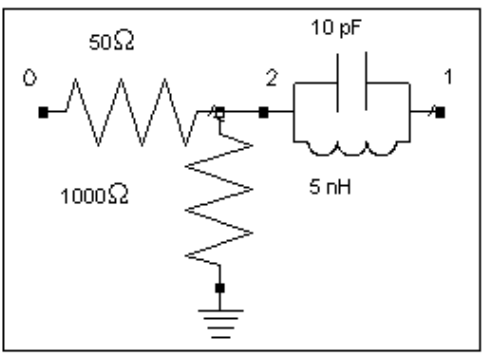


Figure 4-1. Element model for transient analysis

Using Transmission Lines

The `fix_tr` function is called just before transient analysis begins. Its only purpose is to set up ideal transmission lines for the user. Using the `add_tr_tline` function, transmission line pins and physical constants are defined. All four terminals of the transmission line are available to the user (*See* Figure 4-2). Once the transmission line is defined here, time-domain analysis of it is performed automatically without any further action by the user in the `analyze_tr` function.

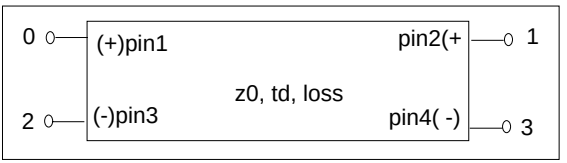


Figure 4-2. Four transmission line terminals

The following code sample places a lossless 50Ω transmission line with a 100 psec delay between pins 0 and 1. An `analyze_tr` function is still required, even though it doesn't do anything—it should simply return `TRUE`.

```
boolean example2_fix (UserInstDef *pInst)
{
   boolean status;
    status = add_tr_tline(pInst, 0, 1, GND, GND, 50.0, 100.0e-12, 1.0);
   return status;
};
boolean example2_tr (UserInstDef *pInst, DOUBLE *vPIn)
{
   return TRUE;
}
```

User-defined P-N Diode Model

This section shows how to extend the P-N diode example created in Chapter 3 for use in a transient model. Only the new code and modifications required to extend this element to a transient model are listed below. The code for this model is available in the example `PNDIODE`. The model shown in Figure 4-3 is used for the `PNDIODE` element.

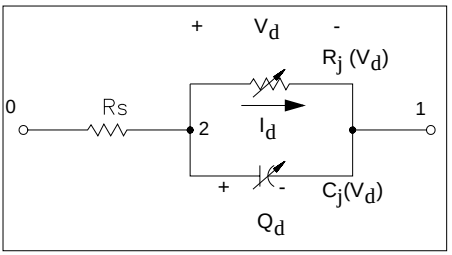


Figure 4-3. PNDIODE element model

Defining the Transient Device

A prototype for the transient analysis function is required and is automatically generated in the PNDIODE_h.c file when the Transient Function check box is selected in the Code Options dialog box.

```
#define ANALYZE_TR_FCN_PTR     analyze_tr
static boolean analyze_tr(UserInstDef *userInst, double *vPin);
```

A UserTranDef structure is defined for transient.

```
#define ANALYZE_TR_DEF_PTR    analyze_tr_def_ptr
static UserTranDefstatic UserTranDef
ANALYZE_TR_DEF_PTR =
{
  NUM_TRANSIENT_INT_NODES,    /* numIntNodes */
  NUM_TRANSIENT_CAPS,     /* numCaps */
  NUM_TRANSIENT_INDS,      /* numInds */
  NUM_TRANSIENT_TLS,       /* numTlines */
  USE_CONVOLUTION,      /* useConvolution */
  ANALYZE_TR_FCN_PTR,       /* analyze_tr */
  FIX_TR,        /* fix_tr */
};
```

In the UserElemDef, a pointer to the DIODE_TR structure is added at the end via a macro.

```
#define ANALYZE_TR_DEF_PTR      analyze_tr_def_ptr
static UserElemDef PNDIODE_ELEMENTS[] =
{
  “PNDIODE”,   /* modelName */
  ..
  &ANALYZE_NL_DEF_PTR,    /* nonlinear structure ptr */
  NULL,                   /* User-defined arb. data structure */
  &ANALYZE_TR_DEF_PTR,    /* transient fcn ptr */
}
```

Transient Analysis Function

The analysis routine diode_nl_iq_gc that was written for the nonlinear model (Refer to Chapter 3, Creating Nonlinear Circuit Elements) can also be used for the transient model. Add to this the contribution of the series resistance and the model is complete. The analyze_tr function that follows calls the diode_nl_iq_gc function for the nonlinear contribution and loads them into the matrix, and then uses add_tr_resistor to include the contribution of the series resistance.

```
static boolean analyze_tr(
  UserInstDef *userInst,
  double      *vPin)
{
    UserParamData *pData = userInst->pData;
    char *pMsg = NULL;
    boolean status;
    double id, qd, gd, capd, rs;

  /* compute the nonlinear portion */
    diode_nl_iq_gc(userInst, vPin, &id, &qd, &gd, &capd);

    status = add_tr_iq(userInst, 2,  id,  qd) &&
           add_tr_iq(userInst, 1, -id, -qd);
    if (status == FALSE) goto END;

   status = add_tr_gc(userInst, 2, 2,  gd,  capd) &&
            add_tr_gc(userInst, 2, 1, -gd, -capd) &&
            add_tr_gc(userInst, 1, 2, -gd, -capd) &&
            add_tr_gc(userInst, 1, 1,  gd,  capd);
    if (status == FALSE) goto END;
  /* series resistance */
    if (AREA_P > 0.0)
      rs = RS_P / AREA_P;
    else
      rs = 0.0;
    status = add_tr_resistor(userInst, 0, 2, rs);

   END:
  if (pMsg)
  {
      (void)sprintf(ErrMsg, "Error: PNDIODE: analyze_tr(%s) -> %s",
     userInst->tag, pMsg);
    send_info_to_scn(ErrMsg);
  }
  return status;
} /* analyze_tr() */
```

The next step is compiling and linking the code. Refer to Chapter 1, Building User-Compiled Analog Models.

Referencing Data Items

Refer to “Referencing Data Items” on page 1-29.

Displaying Error/Warning Messages

Refer to “Displaying Error/Warning Messages” on page 1-30.

Chapter 5: Custom Modeling with Symbolically-Defined Devices

This chapter presents a powerful capability of Advanced Design System: the ability to create a user-defined nonlinear component which can simulate both the large-signal and small-signal behavior of a nonlinear device, without the use of source code.

The *symbolically-defined device* (SDD) is an equation-based component that enables you to quickly and easily define custom, non-linear components. These components are multi-port devices that can be modeled directly on a schematic. You define an SDD by specifying equations that relate port currents, port voltages, and their derivatives. Equations can also reference the current flowing in another device. Once a model is defined, it can be used with any circuit simulator in Advanced Design System. Derivatives are automatically calculated during the simulation.

Before the SDD, the techniques that were available for modeling nonlinear devices were either limited or cumbersome. One technique was to model the device equations using discrete components—usually resistors, capacitors, inductors, and controlled sources. Since most simulators restrict these devices to be linear, this approach could be used to model only the small-signal (AC) behavior of the nonlinear device, and you could not achieve an accurate DC simulation or harmonic balance simulation. A second approach would be to use measured data, typically S-parameters, to model the device, but this approach, too, modeled only small-signal behavior.

The only technique previously available to develop a model that simulated both the large-signal and small-signal behavior of a nonlinear device required writing source code, which was a lengthy task. For example, a typical BJT model would require over 4500 lines of code, and could take an experienced engineer well over a month to write and debug. There also is the requirement that the simulator be linked to your compiled code.

By comparison, the SDD offers a simple, fast way to develop and modify complex models. Equations can be modified easily, and simulation results can be compared to measured data within Advanced Design System.

The SDD can also model high-level circuit blocks such as mixers or amplifiers. By using a single, high-level component instead of a subcircuit of low-level devices, simulations run more quickly. If second- and third-order effects of low-level subcircuits need to be analyzed, the SDD can be modified to develop a more comprehensive implementation of the circuit.

The examples in this chapter start with a simple nonlinear resistor, then more complex devices, like the Gummel-Poon charge-storage model of the bipolar junction transistor, are described. With the techniques used to develop these models, you can develop your own, custom, nonlinear components.

This chapter has the following sections:

- "Writing SDD Equations" on page 5-3 explains how to write the equations that define an SDD.
- "Adding an SDD to a Schematic" on page 5-13 describes how to add an SDD to a schematic and enter equations.
- "SDD Examples" on page 5-17 show how to use SDDs to define a wide range of nonlinear circuit components.
- "Modified Nodal Analysis" on page 5-45 is a discussion of modified nodal analysis and branch equations.
- "Error Messages" on page 5-50 lists SDD error messages and their meaning.

Detailed knowledge of microwave and RF circuit theory and of building and analyzing circuits using Advanced Design System is assumed.

Writing SDD Equations

The symbolically-defined device is represented on the circuit schematic as an *n*-port device, with up to 10 ports. The equations that specify the voltage and current of a port are defined as functions of other voltages and currents. An example of a 2-port SDD is shown here.

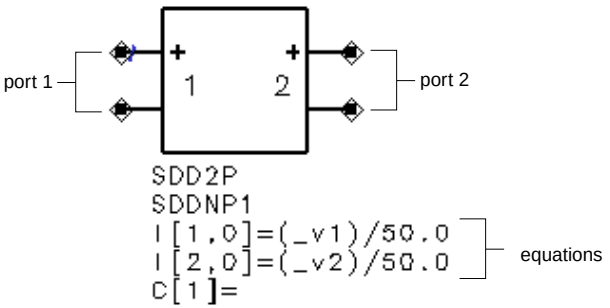


Figure 5-1. The schematic symbol for a two-port SDD

Port Variables

For each port on the SDD, there are voltage and current *port variables*. A variable begins with an underscore, followed by v (for voltage) or i (for current), and the port number. For example, current and voltage variables for port one are `_i1` and `_v1`, respectively. You can rename variables to better suit the device being modeled. In text, v_n and i_n are used to refer to _vn and _in.

By convention, a positive port current flows into the terminal marked +.

Defining Constitutive Relationships with Equations

A well-defined *n* port is described by *n* equations, called *constitutive relationships*, that relate the *n* port currents and the *n* port voltages. For linear devices, the constitutive relationships are often specified in the frequency domain (for example, as admittances), but since the SDD is used to model nonlinear devices, its constitutive relationships are specified in the time domain.

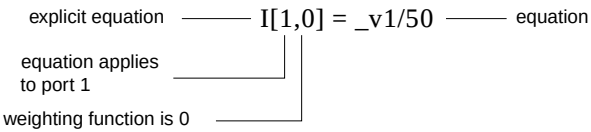
The constitutive relationships may be specified in either *explicit* or *implicit* representations.

Explicit Representation

With the explicit representation, the current at port *k* is specified as a function of port voltages:

$$i_k = f(v_1, v_2, \ldots v_n)$$

An example of an explicit equation is:



In this example, the current at port 1 is calculated by dividing the voltage at port 1 by 50.

Note Each port of the SDD must have at least one equation. For an unused port *n*, apply the equation I[*n*,0] = 0.0 (an open circuit) to the unused port.

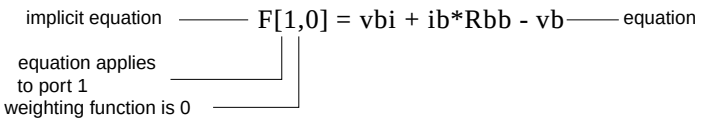
Note Although not often utilized in standard circuit models, the explicit equation defining the i_k port current can actually be a function of any of the port voltages *and* any of the other port currents for ports that are defined with implicit equations. Since port *k* is being defined with an explicit equation, the i_k port variable is not available and so cannot be used to implicitly define i_k.

Implicit Representation

The implicit representation uses an implicit relationship between any of the port currents and any of the port voltages:

$$f_k(v_1, v_2, \ldots v_n, i_1, i_2, \ldots i_n) = 0$$

An example of an implicit equation is:



This equation is part of the Gummel-Poon example.

If you want to use the current variable (*_in*) of a port in another equation, you must define the port using an implicit equation.

A procedure for how to enter equations is in the section "Adding an SDD to a Schematic" on page 5-13.

Explicit Versus Implicit Representations

The explicit representation is a voltage-controlled representation and can implement only voltage-controlled expressions. The implicit representation has no such restriction. It can model equations that are voltage-controlled, current-controlled, or use some other control.

Although implicit equations have no restrictions, explicit equations are more “natural” and more efficient. The explicit representation is more natural simply because many models are expressed in the voltage-controlled form $i = f(v)$. The corresponding implicit equation is $i - f(v) = 0$, which is less intuitive.

Explicit equations use standard nodal analysis, that is, the sum of the currents entering and exiting a node equal zero. Implicit equations use modified nodal analysis, which adds a branch equation and makes i_k available as a variable. For more information on modified modal analysis, refer to the section “Modified Nodal Analysis” on page 5-45.

The explicit representation is more efficient during a simulation because it is a voltage-controlled representation and, therefore, does not create any new variables in the modified nodal equations. With implicit equations, for every port that uses an implicit representation, the port current is appended to the list of branch currents and the port equation is appended to the modified nodal analysis equations. The result is a larger system of equations with a larger number of unknowns (for a discussion of modified nodal analysis and branch equations, see the section “Modified Nodal Analysis” on page 5-45).

In general, you should use the implicit representation only when the explicit representation is insufficient. For example, for a given port *n*, the port current variable *_in* can be used in other equation s only if port *n* is defined with an implicit equation.

Continuity

Many of the circuit-solving algorithms used by the simulator are based on the Newton-Raphson algorithm. Consequently, constitutive relationships should conform to the following:

- The functions must be continuous with respect to *v* and *i*.
- Ideally, the functions should be differentiable with respect to *v* and *i*, but it is not required.
- It is desirable if the derivatives are continuous with respect to *v* and *i*, but this is not necessary, for example, a step discontinuity in the derivative is often acceptable.

An example where these considerations are important is piecewise-defined devices where the constitutive relationship changes depending on the region of operation. The constitutive relationships should be carefully pieced together to ensure continuous derivatives at the region boundaries. An example is given in "Full Model Diode, with Capacitance and Resistance" on page 5-28.

Although continuous derivatives are not requires, if a constitutive relationship does not have continuous derivatives, the simulator may have trouble converging, even at low power levels. If you are having convergence problems with an SDD, continuous derivatives is the first thing to check.

Weighting Functions

A *weighting function* is a frequency-dependent expression used to scale the spectrum of a port current.Weighting functions are evaluated in the frequency domain.

There are two predefined weighting functions. Weighting function 0 is defined to be identically one. It is used when no weighting is desired. Weighting function 1 is defined as *jw* and is used when a time derivative is desired.

You can define other weighting functions, starting with the number 2. Weighting functions must be defined in the frequency domain. Weighting functions can, for example, correspond to time delay or to a low-pass or high-pass filter. An example of a time delay weighting function is

$$:e^{-j\times\omega\times delay}$$

Be aware that the SDD will be evaluated at DC, so a user-defined weighting function should be well behaved at jω=0. For example, you might want to use a weighting function of 1/jω to perform time integration, but this will cause a divide-by-zero error at DC.

A procedure for how to enter weighting functions as part of an SDD definition is in the section “Defining a Weighting Function” on page 5-16.

Weighting Function Example

To understand how the weighting functions are used, this example outlines the steps taken to evaluate the port current of an SDD during a harmonic balance simulation.

For simplicity, consider a one-port SDD with an explicit representation for port one:

```
I[1,1] = f(_v1)
```

where *f* is some nonlinear function.

During a harmonic balance simulation, the simulator supplies the SDD with the spectrum $V_1(\omega)$ of the port voltage and asks the SDD for the spectrum $I_1(\omega)$ of the corresponding port current. To evaluate the current, the SDD performs four steps:

$$V_1(\omega) \Longrightarrow v_1(t) \Longrightarrow \hat{i}_1(t) \Longrightarrow \hat{I}_1(t) \Longrightarrow I_1(\omega)$$

1. Perform an inverse Fourier transform on the voltage spectrum $V_1(\omega)$ to obtain a (sampled) time waveform $v_1(t)$.
2. Evaluate the nonlinearity *f* point by point along the time waveform. The result is the (sampled) time waveform $\widehat{i}_1(t)$.
3. Perform a Fourier transform on the time waveform to obtain its spectrum $\widehat{I}_1(\omega)$.
4. Scale the components of this spectrum using the weighting function to obtain the spectrum $I_1(\omega)$ of the port current.

Note The nonlinearity is evaluated in the time domain. The weighting function is evaluated in the frequency domain.

Since multiplication by $j\omega$ in the frequency domain is equivalent to time differentiation in the time domain, in this example, the current is:

$$i_1(t) = \frac{d}{dt} f(v_1(t))$$

You will see this result used in "Nonlinear Capacitors" on page 5-25 and "Nonlinear Inductors" on page 5-32, where the weighting function 1 is used to implement nonlinear capacitors and inductors.

Controlling Currents

Not only can the equations for an SDD be written in terms of its own port voltages and currents, an SDD can also be set up to reference the current flowing in another device. The devices that can be referenced are limited to either voltage sources or current probes in the same network. For instructions on how to define a controlling current, refer to the section "Defining a Controlling Current" on page 5-15 An example appears in "Controlling Current, Instantaneous Power" on page 5-34.

Specifying More than One Equation for a Port

It is possible to specify more than one expression for a port, but they must be either all implicit or all explicit expressions. And, each port must have at least one equation. When more than one expression is given for a port, the SDD calculates a separate spectrum for each expression. Each spectrum is weighted by the weighting function specified for that expression. The SDD then sums up the individual spectra to get the final spectrum. Explicit and implicit examples follow.

Explicit Cases

The two SDD equations

```
I[1,0] = f1(_v1)
I[1,0] = f2(_v1)
```

and

```
I[1,0] = f1(_v1) + f2(_v1)
```

are equivalent and implement

$$i_1 = f_1(v_1) + f_2(v_1)$$

The SDD equations

```
I[1,0] = f1(_v1)
I[1,1] = f2(_v1)
```

implement

$$i_1 = f_1(v_1) + \frac{d}{dt} f_2(v_1)$$

Implicit Cases

The two SDD equations

F[1,0] = f1(_v1, _i1)

F[1,0] = f2(_v1, _i1)

and

F[1,0] = f1(_v1, i1) + f2(_v1, _i1)

are equivalent and implement

$$f_1(v_1, i_1) + f_2(v_1, i_1) = 0.$$

In the case of an implicit representation, if there is only one expression for a port or, equivalently, more than one expression for a port but all the expressions use the same weighting function, do not use a weighting function other than 0. To see this, assume that in the previous example the weighting function is not weighting function number 0 but is the user-defined function H(ω). Then in the frequency domain, the implicit equation becomes

$$H(\omega)F_1(V_1(\omega), I_1(\omega)) + H(\omega)f(V_1(\omega), I_1(\omega)) = 0$$

which is equivalent to

$$F_1(V_1(\omega), I_1(\omega)) + F_2(V_1(\omega), I_1(\omega)) = 0$$

Here, upper-case letters are used to indicate frequency-domain values, and this assumes that the weighting function does not evaluate to zero at a frequency of interest.

You would want to use a weighting function other than 0 with an implicit representation when two or more expressions are used for a port and different weighting functions are used by the expressions. For example, the SDD equations in this example:

F[1,0] = f1(_v1,_i1)

F[1,1] = f2(_v1, _i1)

implement

$$f_1(v_1, i_1) + \frac{d}{dt} f_2(v_1, i_1) = 0$$

Using an SDD to Generate Noise

An SDD can generate noise only for AC and S-parameter simulations. If you want to add 1/f noise to a current source, consider using a standard current noise source and set its value with an equation so it is a function of frequency:

In = 1e-12 + 1e-6/(freq+1)

In the denominator, the 1 is added so that the equation is not divided by zero when freq=0.

Summary

- The SDD is an *n*-port device.
- For port *n*, the voltage is denoted _vn. The current is denoted _in. Positive current flows into the terminal marked +.
- The *explicit representation* is useful for voltage-controlled nonlinearities:

  $i = f(v)$

- The *implicit representation* is useful for the general nonlinearity:

  $f(i,v) = 0$

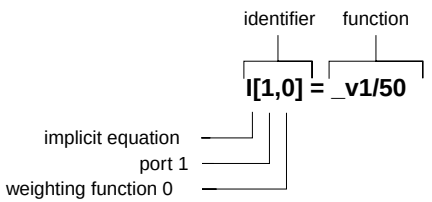
- *Weighting functions* are used to give a frequency weighting to a spectrum. Weighting function number 0 corresponds to no (that is, unity) weighting. Weighting function number 1 corresponds to $j\omega$ and is used to implement a time derivative.
- SDD equations can reference the current flowing voltage sources or current probes in the same network.
- When more than one expression is given for a port, each expression is evaluated, converted into a spectrum, and weighted separately from the others. The resulting spectra are added together to get the final spectrum.
- An SDD can generate noise only for AC and S-parameter simulations.

Adding an SDD to a Schematic

SDDs can be added to a schematic in the same way as other components are added and connected to a circuit. This section describes the mechanics of adding an SDD component to a schematic and defining it.

To add an SDD:

1. From the Component Palette List, choose **Eqn-based Nonlinear**.
2. Select the SDD with the desired number of ports, add it to the schematic, and return to select mode.
3. Double-click the SDD symbol to edit the component.
4. The equations that define the SDD are entered as parameters in the Select Parameters list. The left side of an equation identifies the type of equation, the port it is applied to, and the weighting function



Select the equation you want to edit. (Note the buttons below the list to add, cut, and paste equations as necessary.)

5. Under Parameter Entry Mode, specify the type of equation: **implicit**, or **explicit**. For more information on the types of equations, refer to the section "Defining Constitutive Relationships with Equations" on page 5-4.

6. In the Port field, enter the number of the port that you want the equation to apply to.
7. In the Weight field, enter the weighting function that you want to use. Predefined weighting functions are 0 (the equation is multiplied by 1) and 1 (the equation is multiplied by *jw*). For more information on weighting functions, refer to the section "Weighting Functions" on page 5-8. For information on the procedure for adding a different weighting function to an SDD, refer to the section "Defining a Weighting Function" on page 5-16.
8. In the Formula field, enter the equation. For long equations, click **More** for a larger entry area.
9. Click **Apply** to update the equation.
10. Add and edit other equations for other ports as desired.
11. Click **OK** to accept the changes and dismiss the dialog box.

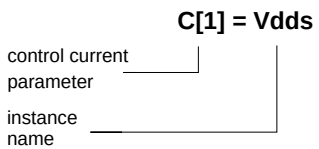
Defining a Controlling Current

The equations for an SDD can be written in terms of the current flowing in another device. For example, you can use the current flowing through a voltage source as part of an SDD equation. You can specify only the current through devices that are either voltage sources or current probes as control currents, and they must be in the same network as the SDD. To specify a current as a control current, you enter the instance name of the device in the *C[]* parameter of the SDD. For example, to use the current flowing through a voltage source called *SRC1*, you would set the current parameter *C[1]* to *SRC1*. The SDD equations use the variable *_c1* to refer to this current.

To define a controlling current:

1. Double-click the SDD component to open the Edit Component dialog box.
2. Select C[1]= in the Select Parameters list.
3. Choose *String and Reference* as the parameter entry mode; *File based* should not be used. In the C[Repeated] field, type the instance name of the device.

An example of a parameter definition is shown here.



4. To add another controlling current, select C[1] and click **Add**. The parameter C[2] appears in the parameter list. You can define this parameter for another current.
5. Click **Apply** to update the SDD definition.
6. To use the controlling current in an equation, type *_cn* in your SDD equation, for example, **_v2 + _v1*_c1**.
7. Click **OK** to accept the changes and dismiss the dialog box.

Defining a Weighting Function

A *weighting function* is a frequency-dependent expression that is used to scale the spectrum of a port current. Weighting functions are evaluated in the frequency domain. Predefined weighting functions are 0 (the equation is multiplied by 1) and 1 (the equation is multiplied by *jw*). You can define your own weighting functions.

To define a weighting function:

1. Double-click the SDD component to open the Edit Component dialog box.
2. Select any equation in the Select Parameters list.
3. Click **Add**. The new equation is automatically selected.
4. From the Parameter Entry Mode list, choose **Weighting**. Note that an **H** appears on the left side of the equation to denote it is a weighting function.
5. In the Weight field, enter a value greater than 1. Each weighting function must have a unique value.
6. In the Formula field, enter the weighting function.
7. Click **Apply** to update the SDD definition.
8. Click **OK** to accept the changes and dismiss the dialog box.

SDD Examples

This section offers the following detailed examples that show how to use symbolically-defined devices to define a wide range of nonlinear circuit components. The examples include:

You can find these examples in the software under the Examples directory in this location:

Tutorials/SDD_Examples_prj/networks

Nonlinear Resistor

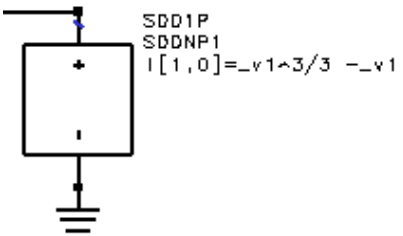
This section describes how to use SDDs nonlinear resistors with a cubic nonlinearity example. This example is under the Examples directory in the following location:

Tutorials/SDD_Examples_prj/networks/Cubic.dsn

The nonlinear two-terminal resistor with constitutive relationship

$$i(v) = v^3/3 - v$$

exhibits a negative resistance for small *v*, and is widely used in the study of oscillation theory. This two-terminal device can be modeled using a one-port SDD, shown below. Since this is a voltage-controlled resistor, the SDD is defined using an explicit equation.



With this setup, note the following points:

- This constitutive relationship specifies the current of port 1, and it is written as a function of the voltage at port 1.
- The Weight field is set to 0 to indicate that the weighting function is identically 1.

Results of DC and harmonic balance simulations on this component are shown in Figure 5-2.

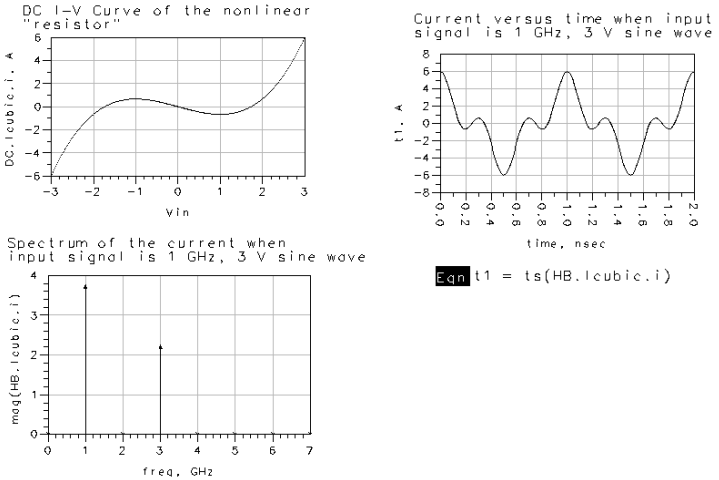


Figure 5-2. Simulation Results For the Nonlinear Cubic Resistor

The data displays show:

- A DC plot of current versus voltage showing the cubic nature of the resistor.
- The spectrum of the resistor current when a 1MHz, 3 V sinusoidal waveform is applied across the resistor. Note that the fundamental and the third harmonic are the only non-zero terms.
- Current versus time with the same waveform applied at the input.

Ideal Amplifier Block

This example is under the Examples directory in the following location:

Tutorials/SDD_Examples_prj/networks/NonlinearAmp.dsn

A simple large-signal model for the gain of an ideal amplifier block can be expressed as

$$v_o = V_s \tanh(A v_i / V_s)$$

where:

- v_i is the input voltage
- v_o is the output voltage
- V_s is the power supply voltage
- A is the gain in the linear region

This relationship has the characteristics that the gain is A for small v_i, and that v_o saturates at $\pm V_s$, as shown in Figure 5-4 (a).

The amplifier is a two-port device, so one more equation is required to specify the constitutive relationship. In the case of Figure 5-3 (a), where the ideal amplifier has infinite input resistance and zero output resistance, you could use the above equation and the equation $i_i = 0$ to define the constitutive relationship.

To model the amplifier as shown in Figure 5-3 (b), with finite input resistance R_i and non-zero output resistance R_o, the equations will be different. The SDD in this example is based on this model.

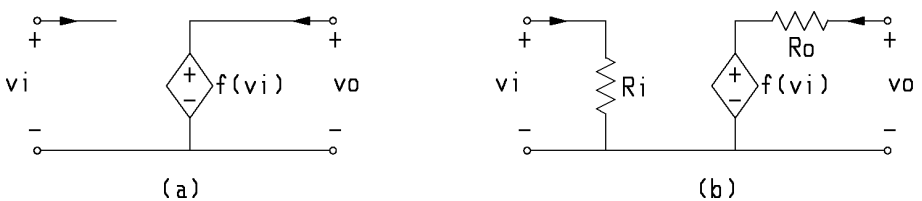


Figure 5-3. Equivalent Circuit Model for an Ideal Saturating Amplifier

Current through the input resistance R_i can be expressed as:

$$i_i = (v_i / R_i).$$

You could use this equation directly as the equation for port 1, but then it would be impossible to set $R_i = \infty$. So, rewrite the explicit equation for port 1 using input conductance G_i instead:

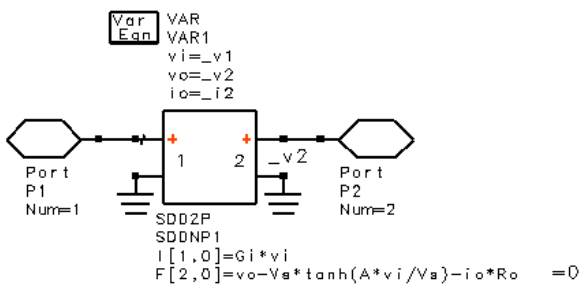
$$i_1 = G_i \cdot v_i$$

For port 2, the non-zero output resistance R_o is included in the model by adding a term to the equation $f(v_i) = V_s \tanh(A v_i / V_s)$ to account for the voltage drop across the output resistance:

$$v_o = V_s \tanh(A v_i / V_s) + i_o R_o$$

Note We can use the port 2 current in this equation because the equation for port 2 is an implicit equation. Recall that when the equation for port n is implicit, the simulator appends the current through port n to the list of unknowns and, therefore, the value of *_in* is available.

This model of an ideal amplifier as two-port SDD with the mixture of explicit and implicit equations is shown below.



Note the following points:

- There are several parameters whose values are set by the user then passed to the device: G_i (input conductance), A (gain), V_s (saturated output voltage), and R_o (output resistance).
- *_v1*, *_v2*, and *_i2* are assigned to variables (*vi*, *vo*, and *io*, respectively), and the variables are used in the SDD equations.
- The final form of the implicit equation for port 2 is written so that it equates to zero.

The SDD is simulated in the design *TestAmp.dsn*. DC and harmonic balance simulation results are shown in Figure 5-4.

- The first plot is a DC plot of v_o versus v_i.
- The second plot is harmonic balance results showing output power and gain as the amplifier saturates.

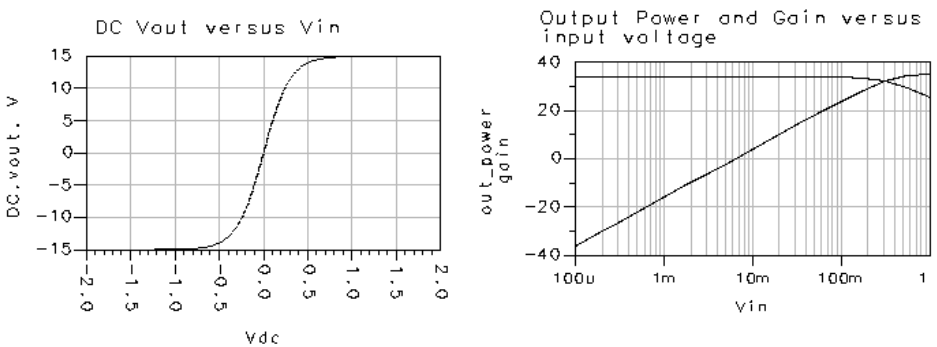


Figure 5-4. Simulation Results for the Ideal Saturating Amplifier

Ideal Mixer

This example is under the Examples directory in the following location:

Tutorials/SDD_Examples_prj/networks/IdealMixer.dsn

The equivalent circuit for an ideal mixer is shown in Figure 5-5.

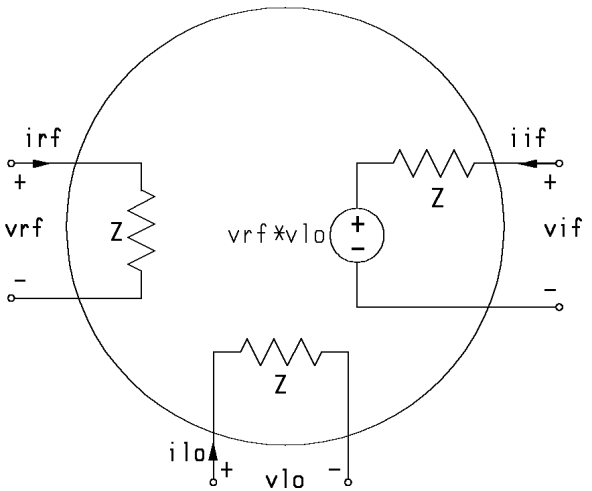


Figure 5-5. Equivalent Circuit for an Ideal Mixer

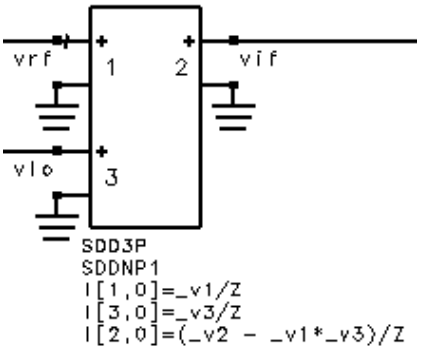
The ideal mixer is a three-port device, so three equations are required to define its constitutive relationship. Based on the circuit above, the following three equations can be used to represent the current at each port:

$i_{rf} = v_{rf}/Z$

$i_{lo} = v_{lo}/Z$

$i_{if} = (v_{if} - v_{rf}v_{lo})/Z$

These equations are voltage-controlled and can be implemented using explicit SDD equations. The SDD is shown next.



In this setup *_v1*, *_v2*, and *_v3* were used in the equations. Each port has a named node, so the voltages will appear in the data display.

RF-LO-IF Example

An RF input of 1V at 3 GHz and an LO input of 1V at 4 GHZ yields an IF output of 0.25V at 1 GHz and 7 GHz, provided the IF output is matched (terminated in Z). The one scaling down by a factor of two comes from the ideal mixing process, while the other comes from the voltage being split over the two Zs.

Figure 5-6 shows the results of a harmonic balance simulation of the mixer. It shows the amplitude modulation effects in the time waveform of $v_{if.}$ For this simulation, v_{rf} is a sinusoid at 100 MHz with a DC offset, and v_{lo} is a sinusoid at 2 GHz.

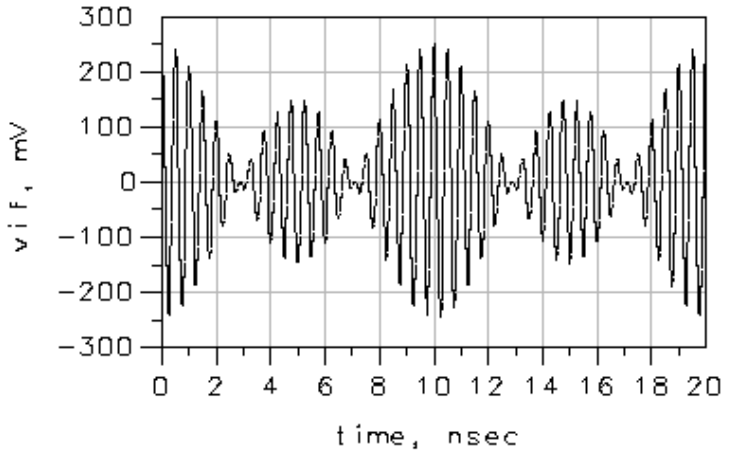


Figure 5-6. Simulation Results for the Ideal Mixer

Nonlinear Capacitors

So far, all of the examples have dealt with nonlinear resistors. This section describes nonlinear capacitors.

A nonlinear, voltage-controlled capacitor is defined in terms of its charge-voltage, or *q-v*, relationship

$$q = Q(v)$$

For example, the *q-v* relationship for a linear two-terminal capacitor is

$$q = Cv$$

which, when differentiated with respect to time, yields the more familiar capacitor equation

$$i = C(v)\frac{dv}{dt}$$

To use the SDD to model a nonlinear voltage-controlled capacitor, note that given a nonlinear charge Q(v), the current is

$$i = \frac{d}{dt}Q(v)$$

This is a voltage-controlled expression for the current. It differs from the constitutive relationship of a voltage-controlled resistor because it contains a time derivative.

The time derivative is implemented in the SDD by specifying weighting function number 1. Weighting function number 1 is predefined as $j\omega$ which is the frequency-domain version of the time derivative.

Obtaining Charge From Capacitance

Often the equation for a nonlinear capacitor is specified not in terms of charge, but in terms of a nonlinear capacitance *C(v)* where

$$i = C(v)\frac{d}{dt}$$

Given this representation, the charge function is obtained by integrating the capacitance

$$Q(v) = \int_{v} C(\hat{v})\, d\hat{v} + Q_o$$

where we have explicitly included the arbitrary constant of integration Q_0.

If for some reason, the charge cannot be calculated, then the alternative technique presented in "Alternative Implementation of a Capacitor" on page 5-47 can be used to implement the capacitor.

Multi-port Capacitors

A nonlinear voltage-controlled two-port capacitor is usually defined by a capacitance matrix

$$C(v_1, v_2) = \begin{bmatrix} C_{11}(v_1, v_2) & C_{12}(v_1, v_2) \\ C_{21}(v_1, v_2) & C_{22}(v_1, v_2) \end{bmatrix}$$

The capacitor currents are given by

$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = C(v_1, v_2) \frac{d}{dt} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

The charge for a two-port capacitance is defined as the function $Q(v_1,v_2)$ such that $C(v_1, v_2)$ is the derivative (that is, Jacobian) of $Q(v_1,v_2)$. It follows that $Q(v_1,v_2)$ exists if and only if

$$\frac{\partial C_{11}}{\partial v_2} = \frac{\partial C_{12}}{\partial v_1}$$

and

$$\frac{\partial C_{21}}{\partial v_2} = \frac{\partial C_{22}}{\partial v_1}$$

If $Q(v_1,v_2)$ does not exist, then the technique presented in "Alternative Implementation of a Capacitor" on page 5-47 can be used to implement the capacitor.

Full Model Diode, with Capacitance and Resistance

This example is under the Examples directory in the following location:

Tutorials/SDD_Examples_prj/networks/SDD_Diode.dsn

Capacitance. The junction capacitance of a reverse-biased pn diode may be written as

$$C_r(v) = C_o\sqrt{\frac{V_o}{V_o - v}} \qquad v < V_o$$

The subscript *r* signifies reverse bias.

To develop this expression into an equation that can be used in an SDD, you integrate $C_r(v)$ with respect to *v* to get an expression for the charge:

$$Q_r(v) = -2C_o\sqrt{V_o(V_o - v)} \qquad v < V_o$$

where the arbitrary constant of integration is chosen so that $Q_r(V_o) = 0$.

There is a limitation to this equation because it is valid only for $v < V_o$. Though it is useful in applications where the diode is always reverse biased (for example, a varactor diode), it is not suitable for a general harmonic-balance analysis (or a DC analysis, for that matter) where the bias voltage may exceed V_o.

A better diode model has the charge model extended into the forward-biased region, plus resistance. Capacitance is described next, followed by resistance and the SDD implementation. Besides yielding a valuable result, this example also highlights some useful techniques for ensuring the continuity of charge and its derivative.

To increase the range of operation of the model, you can extend the capacitance into the region $v > V_o$ using a linear extrapolation. To do this, choose α such that $0 < \alpha < 1$.

Let the previous $C_r(v)$ equation be valid for $v < \alpha V_o$, and for $v < \alpha V_o$ use

$$C_f(v) = C_r(\alpha V_o) + C'_r(\alpha V_o)(v - \alpha V_o)$$

where:

- $C'_r(v)$ is the derivative of $C_r(v)$ with respect to *v*
- The subscript *f* signifies forward bias
- C_f is a linear extension of C_r that matches the value and slope of C_r at $v = \alpha V_o$

This definition of C_f ensures that, when joined with C_r, the capacitance and its derivative are continuous. The boundary between reverse and forward bias is chosen to be αV_o instead of V_o because the slope of C_r at V_o is infinite.

The next step is to integrate $C_f(v)$ to obtain

$$Q_f(v) = (v-\alpha V_o)(C_r(\alpha V_o) + C'_r(\alpha V_o)(v-\alpha V_o)/2) + Q_r(\alpha V_o) \qquad v \geq \alpha V_o .$$

where the constant of integration is chosen so that $Q_f(\alpha V_o) = Q_r(\alpha V_o)$. This equation can be rewritten as

$$Q_f(v) = \frac{C_o}{\sqrt{1-\alpha}}\left(v - \alpha V_o + \frac{(v-\alpha V_o)^2}{4V_o(1-\alpha)}\right) + Q_r(\alpha V_o) \qquad v \geq \alpha V_o .$$

The overall expression for the junction charge is given as

$$Q(v) = \begin{cases} Q_r v \text{ if } v < \alpha V_o \\ Q_f v \text{ if } v \geq \alpha V_o \end{cases}$$

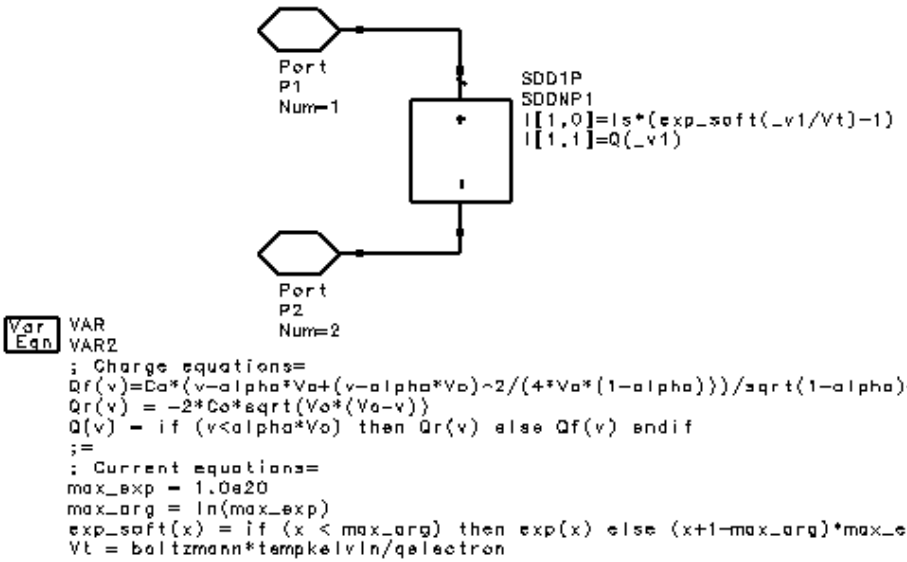
Note $Q(v)$ and its derivatives are guaranteed continuous due to the definition of $C_f(v)$ and due to the choice of the constant of integration for $Q_f(v)$.

Resistance. The equation for the resistive behavior of a pn junction is the ideal diode equation

$$i = I_s(\exp(v/V_T) - 1).$$

Thus, total diode current has two components, one from the ideal diode equation and one from the charge. This is handled in the SDD by specifying two equations for the current of port one, one using weighting function number 0 and the other using weighting function number 1.

Implementation. The SDD implementation is shown next.



Note the following points:

- The current in the diode is based on two SDD equations:
  - The first equation models the resistive behavior of the diode. It uses expressions listed in the Var Eqn component under Current equations. These include the variables *max_exp*, *max_arg*, the function *exp_soft(x)*, and the variable *Vt*. They determine what value *Is* is multiplied by. *exp_soft* is the soft exponential function and is used to prevent overflow problems when taking the exponent of a large number. It is the same as a normal exponential except it becomes a linear extrapolation when its argument is such that the normal exponential would exceed *max_exp*.
  - The second equation models the charge. It uses the expressions listed in the Var Eqn component under Charge equations. The value of *_v1* is passed to the function *Q(v)*, where it is evaluated and the result is returned to the SDD. There are several parameters with user-defined values, which also enter into the calculations: *Is* (), *Co* (), *Vo* (), and *alpha* () (these value are passed from *TestDiode.dsn*).

- A weighting function is used in the second SDD equation. It is important to understand how the weighting function is used by the SDD and is reviewed here.
  - The spectrum for the port voltage *_v1* is inverse Fourier transformed into the time domain.
  - The constitutive relation (in this case, *-2*C0*sqrt(V0*(V0-vv)*) is evaluated point-by-point in the time domain.
  - The resulting waveform (which is the charge for port one) is Fourier transformed into the frequency domain.
  - The weighting function (in this case, *jw*) is applied in the frequency domain. The result is the spectrum of the port current *_i1*.
- When two explicit equations are specified for a single port, the SDD calculates a spectrum representing the (weighted) result of the first equation, calculates a spectrum representing the (weighted) result of the second equation, and then sums the two spectra to get the final spectrum for the port current.

The SDD is simulated in the design *TestDiode.dsn*. This design uses the diode capacitance as the *C* in an *RC* circuit. It also allows the independent adjustment of the diode bias voltage. Figure 5-7 shows the frequency response of the *RC* circuit as the bias voltage is varied fro -1 to 2 V.

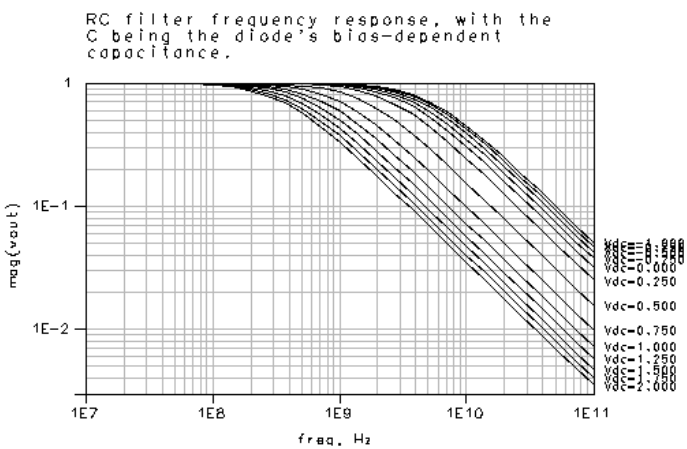


Figure 5-7. Full Varactor Diode Model Results with C_0 = 1 pF, V_0 = 0.65V, and α = 0.7

Nonlinear Inductors

A nonlinear current-controlled inductor is defined in terms of its flux-current, or ϕ-i, relationship

$$\phi = \Phi(i).$$

For example, the ϕ-i relationship for a linear two-terminal inductor is

$$\phi = L\,i$$

which, when differentiated with respect to time, yields the more familiar inductor equation

$$v = L\frac{di}{dt}.$$

To model a current-controlled nonlinear inductor, differentiate

$$\phi = \Phi(i)$$

with respect to time to obtain

$$v = \frac{d}{dt}\Phi(i)$$

which can be rewritten as

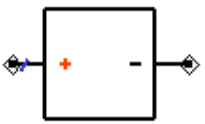
$$v - \frac{d}{dt}\Phi(i) = 0.$$

This expression can be implemented using an implicit representation.

For example, the SDD implementation for the nonlinear inductor specified by

$$\Phi(i) = L_1 i + L_3 i^3$$

is




```
SDD1P
SDD1P1
F[1,0]=v1
F[1,1]=-Phi(i1)
```

```
Var  VAR
Eqn  VAR1
     v1=_v1
     i1=_i1
     Phi(x)=L_1*i + L_3*i^3
```

Note This is a good example of a case when using weighting functions with the implicit representation makes sense.

Obtaining Flux From Inductance

Often the equation for a nonlinear inductor is specified not in terms of flux, but in terms of a nonlinear inductance $L(i)$ where

$$v = L(i)\frac{di}{dt}.$$

Given this representation, the flux function is obtained by integrating the inductance

$$\Phi(i) = \int_i L(\hat{i})\, d\hat{i} + \Phi_0$$

where we have explicitly included the arbitrary constant of integration ϕ_0.

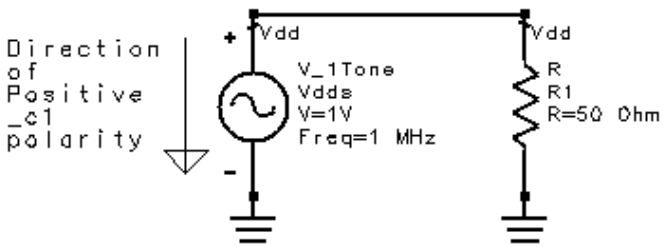
Controlling Current, Instantaneous Power

This example is under the Examples directory in the following location:

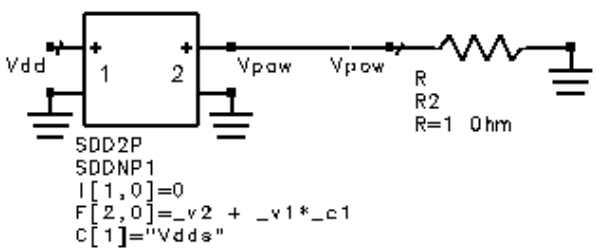
Tutorials/SDD_Examples_prj/networks/RemCC.dsn

This example illustrates how to use a current as part of an SDD equation, where the current is from another device in the circuit. For more background on controlling currents and how to implement them, refer to "Controlling Currents" on page 5-10 and "Defining a Controlling Current" on page 5-15.

In this example, an SDD is used to calculate the instantaneous power dissipated through resistor R1. The circuit containing R1 is shown here.



Making the power calculation requires both the voltage across R1 and the current through R1. These values are supplied to the SDD in the following manner:



- The voltage across R1, labeled *Vdd*, is applied to port 1 of the SDD. Note that the current at this port is set to zero.

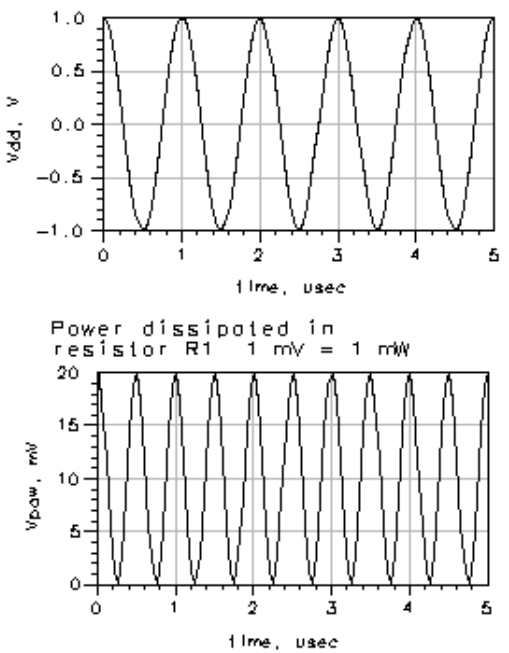
- The current through R1 is specified by using the current through the voltage source *Vdds*, and reversing polarity. Recall that only the current through either a voltage source or current probe can be used as a controlling current. The instance name of the component is used to specify the controlling current, as shown in the SDD illustration. In a more complex circuit, you might consider adding a current probe.
- Although the equation to find power dissipated in R1 is simply Vdd* -_c1, it must be written in a form that is suitable for the SDD. The first step is to substitute _v1 for Vdd. Then note that if:

  _v2 = -_v1*_c1

and by using an implicit equation, the equation

  _v2 + _v1*_c1

can be used to define port 2 of the SDD. Then use a named node (*Vpow*) to save the power to the dataset.The graphs of *Vdd* and the instantaneous power *Vpow* are shown below.



Gummel-Poon BJT

This example is under the Examples directory in the following location:

Tutorials/SDD_Examples_prj/networks/GumPoon.dsn

Figure 5-8 shows the equivalent circuit model for the Gummel-Poon bipolar junction transistor (BJT). The associated current and capacitance equations follow.

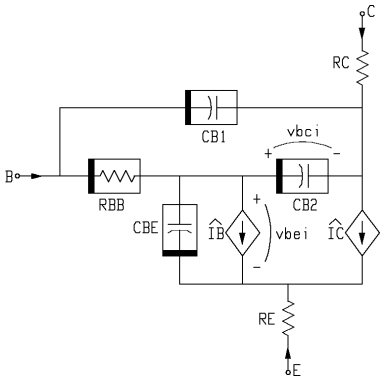


Figure 5-8. Equivalent Circuit Model for the Gummel-Poon BJT

Current Equations

$$\hat{i}_c = \frac{J_s}{Q_b}\left(e^{\left(\frac{v_{bei}}{N_f V_t}\right)} - e^{\left(\frac{v_{bci}}{N_r V_t}\right)}\right) - \frac{J_s}{B_r}\left(e^{\left(\frac{v_{bci}}{N_r V_t}\right)} - 1\right) - J_{lc}\left(e^{\left(\frac{v_{bci}}{N_c V_t}\right)} - 1\right)$$

$$\hat{i}_c = \frac{J_s}{B_f}\left(e^{\left(\frac{v_{bei}}{N_f V_t}\right)} - 1\right) - \frac{J_s}{B_r}\left(e^{\left(\frac{v_{bci}}{N_r V_t}\right)} - 1\right) - J_{le}\left(e^{\left(\frac{v_{bei}}{N_e V_t}\right)} - 1\right) + J_{lc}\left(e^{\left(\frac{v_{bci}}{N_c V_t}\right)} - 1\right)$$

$$R_{bb} = 3 \cdot (rb - rbm) \cdot (\tan(Z) - Z)/(Z \cdot \tan(Z)^2) + rbm$$

where

$$Q_b = \frac{Q_1}{2}(1 + \sqrt{1 + 4Q_2})$$

$$Q_1 = \left(1 - \frac{V_{bci}}{V_{bf}} - \frac{V_{bei}}{V_{br}}\right)^{-1}$$

$$Q = \frac{J_s}{J_{bf}}\left(e^{\left(\frac{V_{bei}}{N_f V_t}\right)} - 1\right) + \frac{J_s}{B_r}\left(e^{\left(\frac{v_{bci}}{N_r V_T}\right)} - 1\right)$$

Capacitance Equations

$$C_{b1} = (1 - X_{cjc}) C_{jc} \left(1 - \frac{V_b - V_{ci}}{V_{jc}}\right)^{-M_{jc}}$$

$$C = \frac{T_r J_s}{N_r V_t} e^{\left(\frac{V_{bci}}{V_T}\right)} + X_{cjc} C_{jc} \left(1 - \frac{V_{bci}}{V_{jc}}\right)^{-M_{jc}}$$

$$C_{be} = \frac{\partial}{\partial v_{bei}} \left(\frac{T_{ff} J_s}{Q_b} \left(e^{\left(\frac{V_{bei}}{N_f V_t}\right)} - 1 \right) \right) + C_{je} \left(1 - \frac{V_{bei}}{V_{je}}\right)^{-M_{je}}$$

where

$$T_{ff} = -TF \left(1 + X_{tf}\, e^{\left(\frac{v_{bci}}{1.44 V_{tf}}\right)} \left(\frac{I_f}{I_f + J_{tf}} \right)^2 \right)$$

$$I_f = J_s \left(e^{\left(\frac{V_{bei}}{N_f V_t}\right)} - 1 \right)$$

Note: Junction capacitances of the form

$$C \left(1 - \frac{v}{V_o}\right)^{-M}$$

change to the form

$$\frac{C}{(1 - F_c)^M} \left(1 + \frac{M}{V_o (1 - F_c)} (v - F_c V_o)\right)$$

when $v > F_c V_o$. Here, $0 < F_c < 1$.

Adding the Nonlinear Base Resistance

In the full Gummel-Poon model, the base resistance R_{bb} is a nonlinear resistance that depends on i_b. When the base resistance is nonlinear, it cannot be modeled by a discrete resistor—it must be included in the SDD equations.

The constitutive relationships are:

$$i_b = (v_b - v_{bi})/R_{bb}$$

$$0 = (v_{bi} - v_b)/R_{bb} + i_b(v_{bei}, v_{bci}) + \frac{d}{dt}(Q_{be} + Q_{b2})$$

$$\hat{i}_c = \hat{i}_c(v_{bei}, v_{bci}) - \frac{d}{dt}Q_{b2}$$

$$i_e = -\hat{i}_b(v_{bei}, v_{bci}) - \hat{i}_c(v_{bei}, v_{bci}) - \frac{d}{dt}Q_{be}.$$

Adding the Split Base-Collector Charge

Now that the nonlinear base resistance has been modeled, adding the split base-collector capacitance is straight-forward. First, modify the equation for Q_{b2} to account for X_{cjc}. Second, insert the equation for Q_{b1}. Finally, add the time derivative of Q_{b1} to i_b and subtract it from i_c:

$$i_b = (v_b - v_{bi})/R_{bb} + \frac{d}{dt}Q_{b1}$$

$$0 = (v_{bi} - v_b)/R_{bb} + i_b(v_{bei}, v_{bci}) + \frac{d}{dt}(Q_{be} + Q_{b2})$$

$$i_c = \hat{i}_c(v_{bei}, v_{bci}) - \frac{d}{dt}(Q_{b1} + Q_{b2})$$

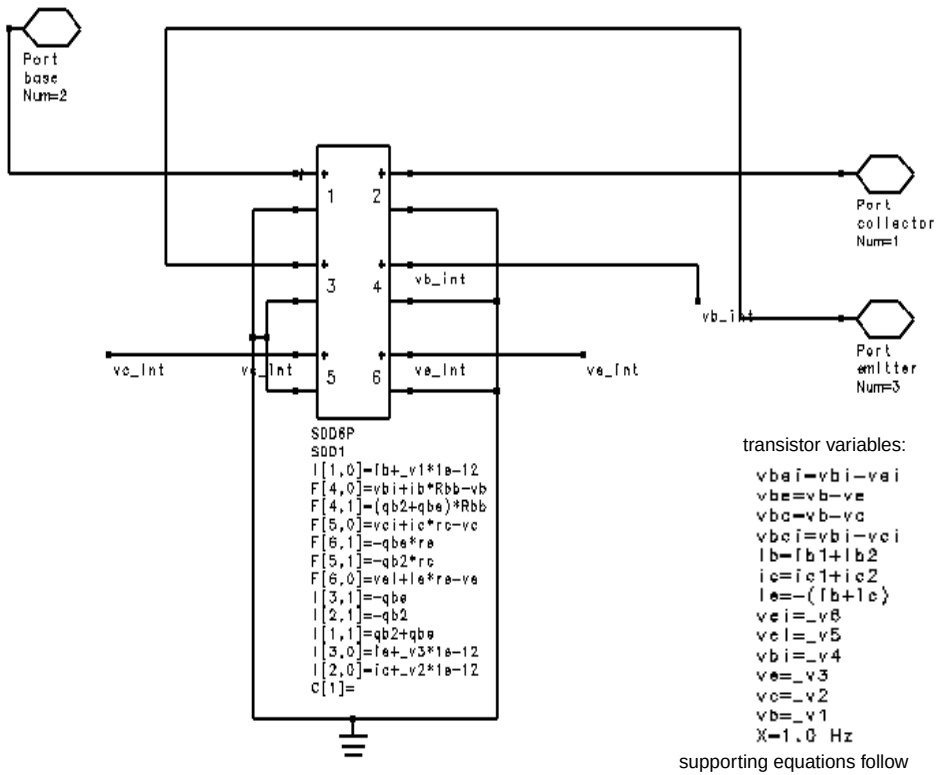
$$i_e = \hat{i}_b(v_{bei}, v_{bci}) - \hat{i}_c(v_{bei}, v_{bci}) - \frac{d}{dt}Q_{be}.$$

SDD Implementation

This implemented SDD is under the Examples directory in the following location:

Tutorials/SDD_Examples_prj/networks/GumPoon.dsn

For optimal viewing, you should open the design. The components and equations are shown below.



Var Eqn VAR
_VAR1

```
;temperature variables and equations=
vje_T=vje
Br_T=br
Eg(t)=eg0-7.02e-4*(t+TzeroC)^2/(1108+(t+TzeroC))
Bf_T=bf
eps=1e-9
Js_T=Js*TempRatio^xti * exp_soft(Eg(Tamb)*((Tj-Tamb)/((Tamb+TzeroC)*vt)))
TempRatio=(Tj+TzeroC)/(Tamb+TzeroC)
TzeroC=273.15
Tj=Tamb
;=
;non-linear base resistance=
Ibb=if (Ib < 1pA ) then 1pA else Ib endif
Z=(-1 + sqrt(144*Ibb/(pi^2*Jrb) + 1))/((24/pi^2)*sqrt(Ibb/Jrb))
Rbb=3*(rb - rbm)*(tan(Z) - Z)/(Z*tan(Z)^2) + rbm

;=
;base-collector charge=
Qbc=tr*Idiode(Vbci, Js_T, nr) - cjc*Vjc_T*((1-Vbcf/Vjc_T)^(1-mJc))/(1-mJc)
Qb2=Qbc
;=
;base-emitter charge=
Iff=Idiode(Vbef, Js_T, nf)
TFF=tf*(1+xtf*(exp(Vbci/(1.44*vtf)*(Iff/(Iff+Itf))^2)))
Qbe=Idiode(Vbef, TFF*Js/QB, nf) - cje*Vje_T*((1-Vbef/Vje_T)^(1-mJe))/(1-mJe)
;=
;base current equations=
Ib2=Idiode(Vbei, Jle, ne) + Idiode(Vbci, Jlc, nc)
Ib1=Idiode(Vbef, Js_T/Bf_T, nf) + Idiode(Vbcf, Js_T/Br_T, nr)
;=
;collector current equations=
Ic2=-Idiode(Vbci, Js_T/Br_T, nr) - Idiode(Vbci, Jlc, nc)
Ic1=Idiode(Vbei, Js_T/QB, nf) - Idiode(Vbci, Js/QB, nr)
;=
;intermediate variables for collector current=
Q1=1/(1 - Vbcf/Vbf - Vbef/Vbr)
Q2=Idiode(Vbei, Js_T/Ibf, nf) + Idiode(Vbci, Js_T/Ibr, nr)
QB=(Q1/2)*(1 + sqrt(1+4*Q2))
;=
;useful general equations=
vt=boltzmann*(Tamb+TzeroC)/qelectron
Idiode(v,Is,n)=Is*(exp_soft(v/(n*vt)) - 1)
exp_max=ln(1e18)
exp_soft(x)=if (x < exp_max) then exp(x) else (x+1-exp_max)*exp(exp_max) end
```

Note the following points.

- Each port has two equations, one for the current and one for the charge.
- The capacitance equations were integrated to obtain charge equations:
  - The integration is simplified for the first term of C_{be} since the first term is a partial derivative, the integration and partial derivative effectively cancel.
  - The integration is simplified for the first term in C_{bc} since the first term is an exponential, and integration of an exponential is another exponential.
  - The other charges are similar in form to the charge given earlier in the section, "Full Model Diode, with Capacitance and Resistance" on page 5-28.
- The *diode()* and *charge()* functions are used to make the equations more readable and to eliminate the duplication of common expressions.
- Except for one difference, the SDD BJT presented here is *identical* to the compiled BJT model built-in to the simulator (in the simulator, the values of V_{je} and V_{jc} are adjusted to reflect the bandgap characteristics of silicon).
- The SDD BJT uses about 55 equations. The built-in BJT model requires over 4500 lines of C code.
- The SDD BJT was written in about one day, and debugged in about one day. The built-in BJT model required about two weeks to write and another two weeks to debug.

Examples Summary

- A *voltage-controlled nonlinear resistor* is described by its *i-v* relation

  $i = I(v)$

- A two-terminal voltage-controlled nonlinear resistor *i = I(v)* is implemented by

  I[1,0] = _v1

- A *general nonlinear resistor* is described by an implicit *i-v* relation

  $f(i, v) = 0.$

- A general two-terminal nonlinear resistor *f(i,v) = 0* is implemented by

  I[1,0] = f(_i1, _v1)

- A *voltage-controlled nonlinear capacitor* is described by its *q-v* relation

  $q = Q(v).$

- A two-terminal voltage-controlled nonlinear capacitor *q = Q(v)* is implemented by

  I[1,1] = Q(_v1)

- A two-terminal voltage-controlled device with resistance *i = I(v)* and charge *q = Q(v)* is implemented by

  I{[1,0] = I(_v1)

  I[1,1] = Q(_v1)

- If a capacitor is specified by a nonlinear capacitance *C(v)* where

  $i = C(v)\frac{dv}{dt},$

then the corresponding charge is given by

$$Q(v) = \int_{v} C(\hat{v})\, d\hat{v} + Q_o$$

where Q_o is the arbitrary constant of integration.

- A *current-controlled nonlinear inductor* is described by its ϕ-i relation

  $\phi = \Phi(i)$.

- A two-terminal current-controlled nonlinear inductor $\phi = \Phi(i)$ is implemented by

  I[1,0] = _v1

  I[1,1] = -phi(_i1)

- If an inductor is specified by a nonlinear inductance $L(i)$ where

  $$v = L(i)\frac{di}{dt},$$

then the corresponding flux is given by

$$\Phi(i) = \int_i L(\hat{i})\, d\hat{i} + \Phi_o$$

where Φ_o is the arbitrary constant of integration.

- SDD models are easier to write and debug than compiled models, but they are less efficient during a simulation.

Modified Nodal Analysis

Advanced Design System uses nodal analysis to form the circuit equations. Nodal analysis is based on Kirchoff's current law (KCL) which states that for each node, the sum of the currents incident to the node is zero.

Suppose a circuit has n+1 nodes and b branches. Let i be the vector of branch currents. Then KCL can be expressed by the equation

$Ai = 0$

where A is an n x b matrix called the *node incidence matrix*. The entries in A are given by

$$a_{ij} = \begin{cases} 1 & \text{if branch j enters node i} \\ -1 & \text{if branch j leaves node i} \\ 0 & \text{otherwise} \end{cases}$$

In nodal analysis, KCL is not applied to the ground node (such an equation yields no independent information) which explains why A has only n rows. If all the devices in the circuit are voltage controlled, that is, if the port currents of each device are completely determined by the port voltages of that device, then the branch current vector i can be written as

$i = g(v)$

where v represents the vector of n node voltages and g is a map from IR^n to IR^b. Substituting this equation into the KCL equation yields the node analysis equation

$G(v) = 0$

where G is a map from IR^n to IR^n defined by $G(v) = Ag(v)$.

When a circuit contains devices that are not voltage controlled (a voltage source or an inductor, for example), it is impossible to write KCL in terms of the node voltages alone—some additional variables must be used. In *modified nodal analysis*, the branch currents of the non-voltage-controlled devices are retained as variables. Thus KCL can be written as

$$\hat{G}(v, i_b) = 0$$

where i_b is the vector of the n_b branch currents of the non-voltage-controlled devices and G is a map from $IR^{n+n_1}{}_1$ to IR^n. Since there are now n equations in $n+n_b$ unknowns, n_b additional equations must be appended to the node equations. These additional equations are the constitutive relationships of the n_b non-voltage-controlled branches

$$\hat{f}(v, i_b) = 0$$

The resulting augmented nodal equations are the modified nodal analysis equations

$$F(v, i_b) = \begin{pmatrix} \hat{G}(v, i_b) \\ \hat{f}(v, i_b) \end{pmatrix} = 0$$

Alternative Implementation of a Capacitor

Suppose you have a nonlinear capacitance that cannot be integrated to get the corresponding charge function. One example is a capacitance that is table-driven from experimentally obtained data. Another case is a two-port capacitor

$$C(v_1, v_2) = \begin{bmatrix} C_{11}(v_1, v_2) & C_{12}(v_1, v_2) \\ C_{21}(v_1, v_2) & C_{22}(v_1, v_2) \end{bmatrix}$$

where there does not exist a charge $Q(v_1, v_2)$ such that $C(v_1, v_2)$ is the Jacobian of $Q(v_1, v_2)$. In these cases, the capacitor can still be implemented using an SDD. Consider the one-port nonlinear capacitance $C(v)$. By definition,

$$i = C(v)\frac{dv}{dt}$$

There is no way to implement this equation directly using an SDD because it involves the product of a derivative. To bypass this problem, create an intermediate variable $dv\_dt = dv/dt$. Then the capacitor is described by the equations

$$i = C(v)\ dv\_dt$$

$$dv\_dt = \frac{dv}{dt}.$$

There is one problem with implementing these equations directly. In the frequency domain, the time derivative of v is

$$dv\_dt = j2\pi fv.$$

Considering harmonic frequencies, *f* can be as high as 500 GHz. With such a large value of *f*, a 1µV change in *v* produces a 3 MV change in *dv_dt*. This high sensitivity can cause convergence difficulties for the system. To eliminate the problem, scale by a nominal frequency value of 1 GHz.

f_{nom} = 1GHz

i = C(v) f_{nom} dv_dt

$$dv\_dt = \frac{1}{f_{nom}} \frac{dv}{dt}.$$

Note that even though *i* is proportional to f_{nom} *dv_dt*, *i* is not overly sensitive to *dv_dt* because f_{nom} is multiplied by *C*(*v*) which is typically on the order of $1/f_{nom}$.

The scaled formulation of the capacitance is implemented by the SDD using the following equations:

I[[1,0] = C(v)*f_nom*dv_dt

F[2,0] = –dv_dt

F[2,1] = v/f_nom

and these VAR equations:

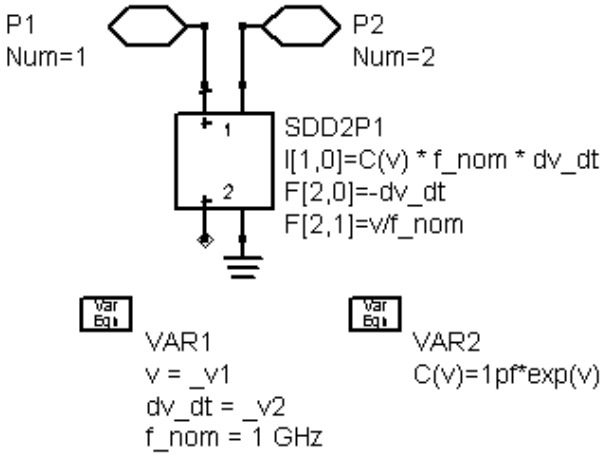
v = _v1

dv_dt = _v2

f_nom = 1 GHz

C(v) = 1pf*exp(v)

This SDD can be found in *Examples/Tutorials/SDD_Examples_prj* as *SDD_cap.dsn*. An alternate implementation, *SDD_cap2.dsn*, can also be found in the project


P1
Num=1
P2
Num=2
1
2
SDD2P1
I[1,0]=C(v) * f_nom * dv_dt
F[2,0]=-dv_dt
F[2,1]=v/f_nom
Var Eqn
VAR1
v = _v1
dv_dt = _v2
f_nom = 1 GHz
Var Eqn
VAR2
C(v)=1pf*exp(v)

Error Messages

If an SDD has not been implemented correctly, it will generate errors. The errors will be similar to the ones listed here.

*Expression error: [*error message*].*

An error has occurred while parsing or differentiating an expression.

h[0] and h[1] are predefined.

Weighting functions 0 and 1 have been redefined. This is not allowed.

Illegal state variable '_in'.

_in has been used, but there are not *n* ports.

Illegal state variable '_vn'.

_vn has been used, but there are not *n* ports.

Improper frequency dependence in sdd 'f' parameters.

One or more of the implicit relationships depends on *freq* or *omega*. Frequency dependence is not allowed.

Improper frequency dependence in sdd 'i' parameters.

One or more of the explicit relationships depends on `freq` or `omega`. Frequency dependence is not allowed.

Port equation cannot be both i and f type.

At least one of the ports has both an explicit and an implicit expression. If more than one expression is used for a port, all expressions for the port must be of the same type, that is, all explicit or all implicit.

Port n is missing a corresponding equation.

No constitutive relationship has been specified for port *n*.

*SYM error: [*error message*].*

An error has occurred while evaluating an SDD expression or its derivative. (SYM is the name of the system symbolic expression handler.)

Chapter 6: Custom Modeling with Frequency-Domain Defined Devices

As CAE plays a larger role in the design cycle of RF and microwave circuits and subsystems, it is important for CAE design systems to satisfy the modeling needs of the engineer at both the device level and the subsystem level. As communication applications continue to increase, it is no longer possible to satisfy all modeling needs with standard, preconfigured models. Thus, Advanced Design System allows users to define their own nonlinear models, in either the time domain or the frequency domain.

For working in the time domain, the symbolically defined device (refer to Chapter 5, Custom Modeling with Symbolically-Defined Devices) enables users to specify nonlinear models directly on the circuit schematic, using algebraic relationships for the port voltages and currents. It works very well for creating many nonlinear device models, but it can be cumbersome for describing the nonlinear, behavioral, frequency-domain operation of the type of subsystems used in RF and microwave communication systems.

To address this need, the *frequency-domain defined device* (FDD) was developed. The FDD enables you to directly describe current and voltage spectral values in terms of algebraic relationships of other voltage and current spectral values. This simplifies development of non-linear, behavioral models that are defined in the frequency domain. The FDD is ideal for modeling a variety of devices, such as modulators and demodulators, phase lock loop components, and more.

The FDD includes capabilities that make it well suited for modeling digital communication subsystems, which often behave in ways that cannot be adequately modeled as time-invariant. Clocked systems, sampled systems, TDMA pulsed systems, and digitally-controlled systems are common, even in the RF and microwave area, and behavioral models must be able to include these effects. So, in addition to its frequency-domain modeling attributes, the FDD also enables you to define trigger events, to sample the voltages and currents at trigger events, and to generate outputs that are arbitrary functions of either the time of the trigger or of the complex spectral voltage and current values at these trigger events.

While the SDD is the user-defined model of choice for modeling at the device and component level where physics dictates that responses are a function of the instantaneous port variables, the FDD is preferable for nonlinear, behavioral modeling in both the frequency and time domains.

Before continuing this chapter, you should be familiar with the SDD. This chapter assumes knowledge of several topics that are presented in the discussion of SDDs, such as port variables and explicit and implicit equations. For more information, refer to Chapter 5, Custom Modeling with Symbolically-Defined Devices.

Signal Models and Sources

To fully understand how FDD models work and what they can do, some understanding of how the simulator models signals in the different simulation analyses is necessary. While the descriptions that follow use voltages, either voltage or current signals can be used.

In *DC* analyses, a node voltage is simply expressed as a constant *V* for all time. Its frequency spectrum is simply an impulse at DC with a value of V.

In *transient* and *convolution* analyses, a node voltage is still a single variable, but it is now a time-varying variable *V(t)*, which can theoretically represent any type of signal from DC up to the Nyquist bandwidth (.5/Tstep). These signals can be periodic, transient, or random signals. The spectrum of this signal can be computed with Fourier transform techniques.

In *harmonic balance* analyses, a node voltage is represented by a discrete spectrum in the frequency domain. This limits the signal types to quasi-periodic signals, and, given memory limitations, to a relatively few number of discrete frequencies. The time-domain waveform can be computed by Fourier transform techniques, based on the equation below.

$$V(t) = \text{real}\left(\sum_{k=0}^{N} V_k e^{j2\pi f_k t} \right)$$

The set of harmonic frequencies is defined by the user entering a set of fundamental frequencies, with an order for each tone. A maximum order parameter is also required for limiting the number of mixing tones that are included in the set of harmonic frequencies. For each of these frequencies, each node voltage has a constant value associated with it, signifying the amplitude and phase of the periodic sinusoid at that frequency.

These frequencies are referenced by fundamental frequency indices, in the following manner:

- Given the indices [*m*,*n*], the corresponding frequency is *m*freq1 + n*freq2*, where *freq1* and *freq2* are fundamental frequencies.

For example, consider a two-fundamental simulation, with fundamental frequencies *freq1* and *freq2* defined as 1GHz and 900 MHz, respectively. The frequency component at 1 GHz would have indices of [1,0]. The 900 MHz frequency component would have indices of [0,1]. 100 MHz would have an index of [1,-1], and [2,-1] would be one of the intermod terms at 1.1 GHz. Note that [0,0] refers to DC. Indices of [-1,1] reference -100 MHz and its spectral values would be equal to the complex conjugate of those at 100 MHz.

A three-fundamental frequency system requires three indices [*m*,*n*,*o*] to define a unique frequency component.

In *Circuit Envelope* analyses, a node voltage is represented by a time-varying, frequency-domain spectrum. As in harmonic balance, a set of harmonic frequencies is user-defined. But here, the spectral amplitude and phase at each of these frequencies can vary with time, so the signal it represents is no longer limited to a constant sinusoid. Each of these harmonic frequencies is the center frequency of a spectrum; the width of each spectrum is ±.5/Tstep. The bandlimited signal within each of these spectra can contain multiple periodic, transient, or random tones. The actual time-domain waveform is now represented by the following equation.

$$V(t) = \text{real}\left(\sum_{k=0}^{N} V_k(t) e^{j2\pi f_k t} \right)$$

Since each time-varying spectrum $V_k(t)$ can be thought of as a modulation waveform of the center frequency f_k, these are often referred to as *envelopes*. This does not imply that there must actually be a frequency component at the center frequency, see Table 6-1 for examples. Note there are N+1 of these spectra. The one at DC (also referred to as the baseband component) is limited to a bandwidth of 0.5/Tstep and must always be real. The other N spectra have a double-sided bandwidth of 1/Tstep and are usually complex.

Table 6-1. Example Signals for Spectrum around f_k

#	Formula	Description
1	V_k=1	Constant cosine cos(2*pi*f_k*time)
2	V_k=exp(-j*pi/2) or polar(1,-90) or -j	Constant sine sin(2*pi*f_k*time)
3	V_k=A*exp(j*(2*pi*f_m*time+B))	One tone (SSB) A*cos(2*pi*(f_k+f_m)*time+B)
4	V_k=A*exp(j*B); freq=1.1 GHz[1]	Same as (3) (assuming f_k + f_m = 1.1 GHz)
5	V_k=2*cos(2*pi*f_m*time)	Two tone (AM suppressed carrier)
6	V_k=exp(j*2*pi*f_m*time) + exp(-j*2*pi*f_m*time)	Same as (5)
7	V_k=pulse(time,...); freq=fk+fm[1]	Pulsed RF at a frequency of f_k + f_m
8	V_k= -step(time - delay)	A negative cosine wave, gated on at t=delay
9	V_k = (vreal(time)+j*vimag(time))* exp(j*2*pi*f_m*time)	I/Q modulated source centered at f_k+f_m. (vreal(), vimag() user-defined functions)
10	V_k=(1 + vr1) * exp(j*2*pi*vr2)	Amplitude and noise modulated source at f_k. (vr1. vr2 user-defined randtime functions)
11	V_k=exp(j*2*pi*(-f_0 + a0*time/2)*time)	Chirped FM signal starting at f_k-f_0, rate a0

1. *freq* is defined in the paragraph below.

The envelope waveform $V_k(t)$ has many useful properties. For example, to find the instantaneous amplitude of the spectrum around f_k at time t_s, you simply compute the magnitude of complex number $V_k(t_s)$. Similarly, the phase, real, and imaginary values of instantaneous modulation can be extracted by simply computing the phase, real, and imaginary values of $V_k(t_s)$. Note this is only extracting the magnitude of the modulation around f_k. It is not including any of the spectral components of adjacent f_{k-1} or f_{k+1} spectra, even if these spectra actually overlap. If this f_k spectrum has multiple tones inside of it, then this demodulation does include their effects.

This simple technique does not allow demodulating only one of the tones inside this f_k spectrum and excluding the other tones in the f_k spectrum. To accomplish this, the desired tone must first be selected by using an appropriate filter in the circuit simulation. Also note that since the baseband (DC) spectrum represents a real signal and not a complex envelope, its magnitude corresponds to taking the absolute value, and its phase is 0 or 180 degrees.

Defining Sources

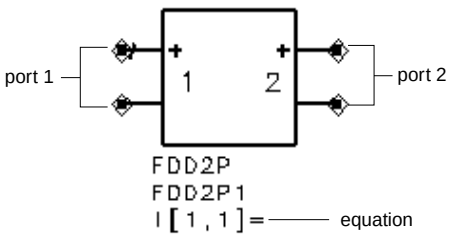
To define a source for Circuit Envelope, you first identify in which spectral envelope the signal belongs. This will typically be the fundamental of one of the frequencies specified in the analysis. Most frequency-domain sources have a single frequency parameter that can be specified. When these sources are used in a harmonic balance or Circuit Envelope simulation, the simulator will determine which of the analysis frequencies is the closest to the source frequency, and if it is close enough, will internally assign it the corresponding set of indices. A Circuit Envelope simulation will also determine the frequency offset from the analysis frequency and automatically shift the signal accordingly. This frequency offset can be up to ±0.5/Timestep. If the source frequency is too far away from any analysis frequency, then its output is set to 0.0 for that analysis and a warning is generated.

Regarding the equations used to define an output from these sources, instead of having to define a fundamental frequency and an SSB frequency offset modulation as in source example 3 in Table 6-1, the simpler format of example 4 is now possible. In addition, these frequency-defined sources are also directly compatible with simple transient analysis.

The Frequency-Domain Defined Device

This section describes the equations and parameters of the FDD. A procedure for adding an FDD to a schematic is in the section “Adding an FDD to a Schematic” on page 6-23. For examples of FDDs developed into models, refer to the section“FDD Examples” on page 6-28.

The frequency-domain defined device is represented on the circuit schematic as an n-port, along with equations specifying the relationships between the spectral port variables. An example of a 2-port FDD is shown here.



By usual convention, a positive port current flows into the terminal marked **+**.

Retrieving Values from Port Variables

The variables of interest at a given port are the port spectral voltages and port spectral currents. Spectral voltages and currents can be obtained using the functions *_sv()*, *_si()*, *_sv_d()*, and *_si_d()*, which are described in Table 6-2, and used in conjunction with equations, which are described in the section “Defining Constitutive Relationships with Equations” on page 6-8.The *_sv()* and *_si()* functions return voltage or current values for a specific port and for a specific frequency. You choose the port by port number, and you choose the frequency using a frequency index. The index is either:

- The index to an FDD carrier frequency and its harmonics
- A set of indices that reference the frequencies of a harmonic balance analysis

For information on FDD carrier frequencies and indexing, refer to the section “Specifying Carriers with the Freq Parameter” on page 6-10. For a description of frequency indices from a harmonic balance analysis, refer to the section “Signal Models and Sources” on page 6-2.

As an example, to access the spectral voltage at port 1 for the second harmonic of the first fundamental frequency, use the function *_sv(1,2)*. Similarly, to access the port *n* current at the frequency with indices [h1,h2,h3], use the function *_si(n, h1, h2, h3)*. Both of these functions return single complex values at each time point, unless the specified envelope is baseband, in which case the value is real. The underscore in the function names is used so as to not conflict with user-defined functions, and to signify that these functions only have meaning within the context of evaluating the FDD device. They cannot be used to directly generate output variables, for example.

The *_sv()* and *_si()* functions return the present value of the specified spectral envelope. For transient, convolution, or Circuit Envelope simulations, it is also desirable to access past values of the spectral port variables. This can be done using the *_sv_d()* and *_si_d()* functions, which are described in Table 6-2. These functions have a delay parameter. For example, to find the value of the port 2 voltage at the [2,-1] intermod frequency 10 µsec ago, use *_sv_d(2, 10us, 2,- 1)*.

Table 6-2. Functions for Accessing Spectral Ports and Currents

Name	Description
_sv(port, findex) †	Returns the spectral voltage for the specified port at the specified frequency index.
_si(port, findex)	Returns the spectral current for the specified port at the specified frequency index.
_sv_d(port, delay, findex)	Returns a past value of spectral voltage for the specified port and time delay at the specified frequency index.
_si_d(port, delay, findex)	Returns a past value of spectral current for the specified port and time delay at the specified frequency index.

† *findex* can refer either to the index of an FDD carrier frequency as defined with the *freq* parameter and its harmonics or a set of indices that reference the fundamental frequencies of a harmonic balance analysis.

The delay values in the *_sv_d()* and *_si_d()* functions can be variables that change during the simulation. However, these delay variables must have their maximum value at time t=0. This is to allow proper initialization of the required history buffers. This criteria can usually be met with an expression such as

fdd_delay = if (time = 0) then max_delay else variable_delay endif.

where *max_delay* is some reasonable value that the *variable_delay* is known to never exceed.

Also, FDDs require that all state variable dependencies that will ever exist must exist at time = 0. For example, the following equation describes a discrete accumulation operation, with a reset to 0 at time = 0:

v[2,0] = if (time = 0) then 0.0 else _sv_d(2,timestep) + _sv(1,0) endif

However, it must be modified to work with the FDD so that both state variable dependencies as well as the maximum delay at time = 0. The following satisfies this criteria by adding an insignificant portion to the time = 0 value.

next_state = _sv_d(2,timestep) + _sv(1,0)

*v[2,0] = if (time = 0) then 0.0 + next_state*1.0e-100 else next_state endif*

Defining Constitutive Relationships with Equations

An unlimited number of equations can be used to define constitutive relationships between the port voltages and port currents. There are two basic types of equations allowed, current equations and voltage equations. Their basic format is:

i[port, findex] = f(_sv(),_sv_d(),_si(),_si_d())

v[port, findex] = f(_sv(),_sv_d(),_si(),_si_d())

The equations can be listed in any order, and more than one equation can be used for a single port, but each port must have at least one equation.

Note the use of indices on the left side of the equations. This is similar to the use of indices in the *_sv()* and other functions that were previously described, they can be can be either the index to an FDD carrier frequency and its harmonics or a set of indices that reference the frequencies of a harmonic balance analysis. Indices are discussed in the sections “Signal Models and Sources” on page 6-2 and “Specifying Carriers with the Freq Parameter” on page 6-10.

In order for a port current to be used on the right side of an equation, at least one voltage equation for that port must be defined. It does not matter which harmonic indices are used for this. Normally, the simulator does not generate current-state variables, only node voltage-state variables. This is sufficient as long as the devices are modeled as voltage-controlled current sources. This is also the most efficient method in terms of speed and memory. However, current-controlled sources can be generated but the simulator requires an additional equation to define this current. The presence of a voltage equation signifies this to the simulator. In general, the voltage equations should only be used when the voltage-controlled current equations are insufficient. Refer to “Modified Nodal Analysis” on page 5-45, for more detail.

While the SDD has truly implicit equations of the form *f(_v1,...,_vn,_i1,...,_in)=0* the FDD does not. However, the equivalent effect can be generated by adding the left side to the right side of the voltage equations. For example,

v[2,1,0]=f(_sv(1,0),_si(2,1,0))+_sv(2,1,0)

has effectively generated the implicit equation *f(...) = 0*. The FDD is different in this respect in order to solve the problem of how to define the voltage at all unspecified spectral frequencies. The above voltage equation also generated a number of additional implied equations that say

v[2,all other harmonic indices] = 0.0.

Note that all equations for the same spectral port variable are effectively added together, so if you also define an equation for another spectral frequency at the same port (i.e., *v[2,0] = vcalc)*, this additional implied equation that sets the voltage to zero is ignored.

For a procedure on how to add current and voltage equations to an FDD, refer to the section "Defining Current and Voltage Equations" on page 6-24.

Continuity

As with any Newton-Raphson based circuit-solving algorithm, the constitutive relationships should be differentiable with respect to each of the specified spectral voltages and currents, and these derivatives should be continuous. Discontinuous derivatives may cause the simulator to have trouble converging.

One possible technique to improve convergence, or to circumvent the above limitations, is to add delay between the input and output using the *_sv_d()* or *_si_d()* functions. If this delay is greater than the simulation timestep, then the derivative information is no longer needed or used. If this delay is acceptable behavior for the model, simulation speed can be improved.

Large step discontinuities in the time-domain functions can also create convergence problems, either taking longer to solve or possibly causing convergence failure. Although having continuous derivatives with respect to time is not as important as having continuous derivatives with respect to the spectral port variables, care should be taken when using abrupt time functions.

Specifying Carriers with the Freq Parameter

The FDD has a repeatable *freq[n]* parameter, which can be used to define one or more carrier frequencies. The *[n]* is used to identify each carrier, for example, *freq[1]=100 MHz, freq[2]= 350 MHz*. If carrier frequencies are defined for the FDD, you can reference them using *_sv()* and related functions in order to collect voltages and currents at carrier frequencies and their harmonics. The syntax to reference them is to set the first index parameter equal to the negative of the carrier frequency number [n]. The optional second index parameter specifies the harmonic of that carrier frequency. For example, if an FDD has defined *freq[3]=800 MHz*, then *_sv(1,-3,3)* specifies the port 1 spectral voltage at the envelope closest to 2.4GHz. If there is no analysis frequency close enough to 2.4 GHz (within 0.5/Timestep for Circuit Envelope), then this function simply returns 0.0 and generates a warning. Note that *_sv(1,3,3)* would be unaffected by the carrier frequency parameters but still refer to the envelope at *3*_freq1 + 3*_freq2*, where *_freq1* to *_freq12* are predefined variables that are set to the fundamental frequencies defined by the analysis.

The FDD *freq* parameter behaves differently than the source *freq* parameter (referred to in the section "Defining Sources" on page 6-5) in that any acceptable offset frequency between the carrier frequency and envelope center frequency is *ignored*. For example, given an envelope analysis with a fundamental frequency of 0.5 GHz, a timestep of 1µsec, and FDD *freq[1] = 500MHz* and *freq[2]=500.1MHz*, then *_sv(1,-1)* and *_sv(1,-2)* would return the same value, the port 1 spectral voltage at 500 MHz. (If timestep is changed to 1msec, then *_sv(1,-2)* would return 0.0.)

Note that it is not possible to reference a mixing product of multiple carrier frequencies. If this is desired, then an additional carrier frequency equal to the desired mixing product frequency must be defined. For an example, refer to the mixer example in the section "Mixer" on page 6-31.

For a procedure on how to add freq parameters to an FDD, refer to the section "Defining Frequency Parameters" on page 6-25.

Creating Output Harmonics

If you are creating a model with the capability to output a large number of harmonics, the *_harm* variable can be used with the FDD to develop a such a model. The *_harm* variable, unlike most of the functions described in this chapter, is not restricted to use in FDDs only.

Using the *_harm* variable, voltage and current source harmonic values can be parametrically defined. Anytime the large signal voltage or current is defined with an expression using *_harm*, the device will automatically index the value of *_harm* from 0 to the maximum value needed for the present analysis. The expression is then re-evaluated for each value of *_harm* to determine the spectral content at a frequency equal to _harm*frequency determined by the parameter indices.

The variable *_harm* is used in the function below, which implements a pulse source:

```
parameters VPEAK=1 V DUTYCYCLE=30 FREQ=1 GHz HARMONICS=16

ivs:CMP79 1 2 freq=Freq v[1]=A(_harm)

 Blackman(n,M) = (0.42 + 0.50*cos(PI*n/M) + 0.08*cos(2*PI*n/M))*step(M-n)

 A(k)=2*VPEAK*DUTYCYCLE/100*sinc(k*pi*DUTYCYCLE/100)*(-1)^k \

     * Blackman(k,Harmonics+1)*step(_harm)
```

The variable *_harm* has no set maximum harmonic limitation. The value of *_harm* will automatically be incremented out to the maximum value available for the present analysis.

This automatic indexing also works in baseband envelope and transient. The variable is incremented until it either reaches 1000 or until its amplitude has become insignificant for several consecutive harmonics.

In non-baseband envelope, the maximum harmonic will also be limited if the source's harmonic falls outside the envelope bandwidth. For example, if the analysis fundamental is 1MHz with a timestep of 1msec (+/-500Hz envelope bandwidth) and the source fundamental frequency is 1MHZ + 100Hz, then the 6th harmonic falls outside the envelope bandwidth and the spectrum is truncated, even if the analysis order is 31. Also, anytime a spectrum is truncated in harmonic balance, it remains truncated even if higher order spectral tones may exist, for example, if another fundamental existed at 10MHz+1kHz, the spectral source would not add energy there even though it is at the 10th harmonic of the source.

Note that the frequency value is determined by the frequency defined in the parameter indices. In the above case, for example, *v[1,2] = A(_harm)* would have defined a pulse waveform whose fundamental frequency is the second harmonic of *Freq*. The equation *v[0,1,2] = A(_harm)* will define a waveform whose fundamental frequency is (*1 *_freq1 + 2*_freq2*) where the *_freqN* variables are the fundamental frequencies defined by the analysis. At the netlist level, multiple different spectrum can be defined in one source, but each one will add the DC (*_harm=0*) value.

Limitations

In general, you should avoid using a fundamental frequency of 0 Hz. The *_harm* parameter is not supported for the small signal spectral parameters.

There is no simulator variable available to determine what the maximum number of harmonics is for a particular case. This can make windowing a little difficult, since a parameter must be used or passed to the model to set the window bandwidth.

Defining an FDD Spectrum

The parametric definition of output spectrum using the *_harm* index also works with the FDD. This example defines a VCO:

```
parameters Kv=1khz FUND=1 Rout=50 ohm PFUND=0 dBm Harmonic=

fdd:CMP2 _NET00005 0 out 0 i[1,0]=0
i[2,f1,f2,f3]=a*Harmonic*exp(j*_harm*b)*step

(_harm)

 a = -dbmtoa(PFUND, a_Rout)

 a_Rout = max(Rout,0.1)

 b = 1000*_sv_d(1,timestep,0)-pi/2

 f1 = if (FUND = 1) then 1 else 0 endif

 f2 = if (FUND = 2) then 1 else 0 endif

 f3 = if (FUND = 3) then 1 else 0 endif

C:CMP3 0 _NET00005 C=1/(2*pi*Kv)

vco_harm:CMP1 vin vout Kv=1khz FUND=1 Rout=50 ohm PFUND=0 dBm
Harmonic=fharm

fharm = sinc(_harm*pi*.25)/sinc(pi*.25)
```

In this example, the user has entered the equation for the spectrum of a 25% duty cycle square wave using the *_harm* index, which can generate as many harmonics as can be supported by the present analysis. The fundamental power is still separately defined so this spectrum is relative to that and the value for *_harm=1* should be 1.0.

This method of harmonic indexing in the FDD is meant primarily for defining multiple spectral outputs dependent on the same spectral input. But the *_harm* index can also be used to change which spectral input is used for each spectral output. An example is *i[2,1]=_sv(2,_harm)/50*, which adds a 50 ohm load at all the harmonics of *fund1*, including DC. Note that this example is to illustrate the capability, it is inefficient compared to using a resistor. Another example is *i[2,1] = _sv(1,0,_harm)/mag(_sv(1,0,1)+tinyreal)/Rout*, which outputs an entire *fund1* spectrum at port 2, based on the port 1 *fund2* spectrum, and limits each spectral component by the *fund2* fundamental magnitude.

Note there is no capability in the FDD to allow automatic outputting of all spectral tones. The *_harm* index is essentially limited to harmonics of the frequency specified by the parameter indices. Additional parameters and equations have to be used to cover additional fundamentals and intermods. An example of this is in the section "Mixer" on page 6-31.

Using Arrays

Sometimes you may want a flexible number of spectral tones, but no simple equation is available for this. Arrays can be used in this case, and unlike the *harmlist* parameter, a separate parameter with separate supporting code is not required. Care must be taken to avoid having the indexes exceed the array bounds, or an error will occur. For this reason, a *length()* function is available to return the length of an array. The following VCO example shows a possible usage. Also, note that the *Harmonics* input parameter is a list of complex numbers representing the relative level of all the desired harmonics, for example, *list(.1, .02_j*.01)*.

```
parameters Kv=1khz Freq=1GHz P=-j*dbmtow(0) Rout=50 Ohm Delay=timestep
Harmonics=

; Y_Port is used as voltage to current converter

Y_Port:CMP1 in 0 _NET00005 0 Y[2,1]=-.001

; This capacitor performs the integration function

C:CMP3 0 _NET00005 C=1/(2*pi*Kv)

; This switch resets the integrating capacitor voltage to 0 at time = 0

ResetSwitch:CMP4 0 _NET00005

; This FDD is a programmable harmonic current source with a phase modulation
input

FDD:CMP2 _NET00005 0 out 0 I[1,0]=0 I[2,-1]=a*exp(j*b)

 I[2,-1]=if (_harm > 1 and _harm <= Hmax+1) then
a*Harmonics[_harm-1]*exp(j*_ha

rm*b)

 else 0.0 endif Freq[1]=Freq


 Hmax = length(Harmonics)

 Pdbm=30.0 + 10.0*log(mag(P) + tinyreal)

 a = -dbmtoa(Pdbm, Rout)*exp(j*phaserad(P))

 b = 1000*_sv_d(1,Delay,0)

R:CMP5 out 0 R=Rout
```

In this case, the model was hard-coded to expect an array. A more general solution might use the above VCO example, but this would require the user to limit the array bounds access, since accessing out of bounds will cause an error.

*vcodata=makearray(0,1,2,.5,.25*j,.125,-.0625,-j*.03125,.015625)*

xyz = if (_harm < length(vcodata) then vcodata[_harm] else 0.0 endif

vco_harm:CMP1 vin vout Kv=1khz FUND=1 Rout=50 ohm PFUND=0 dBm Harmonic=xyz

While this could be used to simplify the *harmlist* implementation, *harmlist* may be more efficient.

Trigger Events

The FDD enables you to define trigger events. Up to 31 triggers can be defined. Anytime the value of the trigger expression is equal to a number other than zero, a trigger event is declared for the corresponding trigger. Each trigger keeps a count of the number times the trigger occurred and the time of its last trigger. The trigger time is defined as the time value of the current simulation point plus the value of the expression. Therefore the value of the expression should normally be the time of the trigger relative to the current time value. The value of this trigger expression should be limited to *-timestep* and *-2*timestep*. This is explained further in the section "Accessing Port Variables at Trigger Events" on page 6-18.

Three built-in functions have been defined to provide access to trigger information, they are described in Table 6-3. Again, the underscore is used as part of the name, signifying that these functions only have meaning within the context of an FDD instance, and are not valid elsewhere.

Table 6-3. Functions available to access trigger information

Name	Description
_to(N)	Returns 1 if trigger N occurred at this time point, else 0
_tn(N)	Returns the accumulated number of trigger N events
_tt(N)	Returns the absolute time in seconds of last trigger N event

Another function is available to detect threshold crossings and to generate the proper trigger expression values, which is shown in Table 6-4. Note that the threshold crossings are based only on the DC (baseband) spectral voltage at the specified port. The actual time crossing is computed based on linear interpolation between adjacent

time points, so the actual accuracy will depend on both the size of the time step and the rate of change of the slope of the signal.

Table 6-4. Function to generate a trigger event

Name	Description
_xcross(P, Vthresh, direction)	Returns 0 if no threshold crossing occurred, otherwise returns its relative time, a value between *(-1 and -2)*timestep*. A threshold crossing occurs if the baseband voltage at port P passes through the value Vthresh in the specified direction. A positive direction number implies a positive edge; a negate number a negative edge; a direction number of 0 implies either positive or negative edge. No hysteresis exists.

For a procedure on how to add trigger parameters to an FDD, refer to the section "Defining Triggers" on page 6-26.

Output Clock Enables

Normally all of the FDD voltages and currents are re-evaluated at every time sample. It is possible, though, to enable the output of a given port to change only when a specified trigger, or a set of specified triggers, occurs. This is done using the clock enable parameter, *ce[n]=value*. *[n]* specifies the port where the clock enable will be applied. *value* is a binary value that is set using the *bin()* function, where the Nth bit corresponds to whether this port should be enabled by the Nth trigger. For example, if you want the output of port *n* to be updated whenever either trigger 1 or trigger 3 occur, you would enter a value of *bin(101)* or *5* for the clock enable parameter. Clock enables can be used when it is necessary to update computed values only at certain time points. Sample-and-holds are one obvious application, refer to the example in the section "Sample and Hold" on page 6-34.

For a procedure on how to add clock enable parameters to an FDD, refer to the section "Defining Clock Enables" on page 6-27.

Accessing Port Variables at Trigger Events

Now that it is possible to generate trigger events at threshold crossings, it is desirable to be able to determine the spectral port voltages and currents at the point in time that this trigger occurred. Linear magnitude and phase interpolation is used to compute values at times between adjacent simulator time points, and again, the accuracy depends on the rate of change of the input envelope waveform. The four functions that are used to do this are described in Table 6-5.

Table 6-5. Functions to access port variables at trigger events

Name	Description
_sv_e(P,N,indices)	Return the port P spectral voltage envelope at the last trigger N time
_si_e(P,N,indices)	Return the port P spectral current envelope at the last trigger N time
_sv_bb(P,N)	Return the total, real voltage of port P at the last trigger N time
_si_bb(P,N)	Return the total, real current entering port P at the last trigger N time

The *_sv_e()* function is very similar to *_sv_d()*, which is described in Table 6-2. By default, though, past history for the *_sv_e()* function is only saved for the last 2 timesteps. Therefore, the event they refer to must have just occurred, and cannot delay back an arbitrary amount of time. If a triggered voltage value is desired at a much later point in time, then it should be sampled and held using a combination of the above functions and the clock enable previously discussed.

All of the spectral port variable functions discussed so far return only the complex value of the single specified envelope. (If the indices are 0, then the real baseband value is returned.) The broadband functions *_sv_bb()* and *_si_bb()* functions, though, perform an inverse Fourier transform of all of the spectral voltages or currents at the specified event time, and return the real value. Note that if this value is computed at every time step, it will generate an aliased, undersampled waveform, since the time step in Circuit Envelope is typically much less than the period of the various envelope center frequencies.

Delaying the Carrier and the Envelope

With exception of the *_sv_bb()* and *_si_bb()* functions, all of the other spectral port variable functions return the envelope information. This is true even with the delayed and event versions. If it is necessary to delay both the envelope and the carrier, then an additional term must be added to account for the carrier phase shift. For example, if the fundamental signal is

Vk(t)*exp(j*2*pi*fc*t)

then

i[2,1]=_sv_d(1,1usec,1)

generates a current equal to

*Vk(t-1μsec)*exp(j*2*pi*fc*t)*

To generate a true coherent delay with the FDD, you would have to modify the equation to

*i[2,1]=_sv_d(1,1usec,1)*exp(-j*2*pi*fc*1usec)*

or to something similar. Of course, if only a fixed delay is desired, there are linear elements that are more suitable for this application than the FDD.

Miscellaneous FDD Functions

There are three remaining functions that are available in the FDD for time-domain operations, and are described in Table 6-6. They were incorporated into the FDD because they required that state history be maintained. The functions correspond to a basic counter and to a linear feedback shift register. These functions are valid only when used in an FDD.

Table 6-6. Miscellaneous FDD Functions

Name	Description
_divn(T,N,N0),	Returns the value of a counter, clocked every time trigger T occurs, decrementing from N to 0. $N0$ is initial time = 0 value.
_lfsr(T, seed,taps)	Returns the value of a linear feed back shift register that is clocked every trigger T. $seed$ is the initial value of the register. $taps$ are the binary weights of the bits that are fed back using modulo 2 math.
_shift_reg(T,M, N, In)	Returns the value of a multi-mode shift register that is clocked every trigger T, has `N` bits, and with an input equal to `In`. M = 0: LSB first, Serial In, Parallel Out M = 1; MSB first, Serial In, Parallel Out M = 2; LSB first, Parallel In, Serial Out M = 3; MSB first, Parallel In, Serial Out

Defining Input and Output Impedances

With the SDD, it is very straight-forward to include the input and output resistances in the basic equations. For example, *i[1]=_v1/50* simply defines a 50 ohm input resistance. This is not as simple with the FDD, since each equation only defines the relationship for a single output spectrum. Thus, *i[1,1,0] = _sv(1,1,0)/50* defines a 50 ohm input resistance, but only for the fundamental spectral envelope. The input resistance for the other spectral components is still infinite, which is the equivalent of being undefined. This becomes more problematic at the output. It is possible to define the output current and output resistance for a single spectral envelope, but to leave the other spectral envelopes undefined. This may create an ill-defined circuit, creating a singular matrix error due to an undefined voltage at certain spectral frequencies. These problems are best circumvented by using actual resistors external to the FDD. Of course, if the resistance for certain spectral envelopes is different from this external value, that difference can be included in the defining spectral port equations.

Compatibility with Different Simulation Modes

The FDD is not fully compatible with all the different circuit analysis modes of Advanced Design System. Since DC, AC, transient, and convolution analyses only define the baseband variables, any use of non-baseband spectral envelopes (harmonic indices not equal to 0) are ignored in these analyses and the voltages and currents for these spectral frequencies are set to 0. Similarly, DC, AC, and harmonic balance analyses are steady-state analyses and time is always equal to 0, so any time-varying functions are evaluated at time=0 and accessing delayed voltages is the same as accessing the present voltage. The concept of generating time trigger events, of course, is valid only in transient, convolution, and Circuit Envelope modes of operation.

Components Based on the FDD

A variety of circuit components in Advanced Design System are based on the FDD. Some of these components are:

- Tuned modulators and demodulators
- Phase lock loop components
- Counter, time, and waveform statistics probes
- Sampler

Many of these models operate on a few (often just one) of the input spectral frequencies, and in turn output just one, or a few, different spectral frequencies. This is consistent with the desired, or measured, primary frequency-domain behavior, and simulations can be performed quite efficiently since all operations are done directly in the frequency domain.

In cases where a model must include second and third-order interactions with other spectral frequency components, and the underlying nonlinearity is an algebraic function of the time-domain voltages and currents, the FDD may become too tedious to generate all of the frequency-domain equations that define the multiple interactions, and a broadband model (which can be developed using the SDD) may be the preferred model.

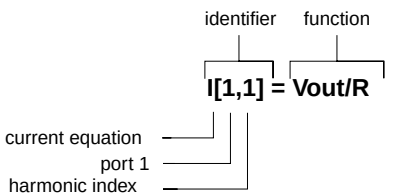
The FDD spectral models, in general, will not function with AC and transient analyses. These limitations are noted where the components are documented in the *Circuit Components* manuals.

Adding an FDD to a Schematic

FDDs can be added to a schematic in the same way as other components are added and connected to a circuit. This section describes the mechanics of adding an FDD component to a schematic and defining it.

To add an FDD:

1. From the Component Palette List, choose **Eqn-based Nonlinear**.
2. Select the FDD with the desired number of ports, add it to the schematic, and return to select mode.
3. Double-click the FDD symbol to edit the component.
4. FDD parameters are entered in the Select Parameters list. The parameter is on the left side of the equation. It identifies the type of parameter, the port it is applied to, and, where appropriate, the harmonic index



Select the parameter you want to edit. (Note the buttons below the list to add, cut, and paste parameters as necessary.)

5. Under Parameter Entry Mode, specify the type of parameter to be defined: **current**, **voltage**, **frequency**, **trigger**, or **clock enable**. Instructions for defining each type of parameter follow.
6. Once a parameter is defined, click **Apply** to update.
7. Add and edit other parameters as desired.
8. Click **OK** to accept the changes and dismiss the dialog box.

Defining Current and Voltage Equations

Current and voltage equations are the two basic types of equations for defining constitutive relationships between the port voltages and port currents. For more information about these equations, refer to the section “Defining Constitutive Relationships with Equations” on page 6-8.

To define current or voltage equations:

1. Double-click the FDD component to open the Edit Component dialog box.
2. By default, a current equation appears in the Select Parameters list. Select this equation.
3. From the Parameter Entry Mode list, choose either **Current** or **Voltage**. For current equations, an I appears on the left side of the equation; for voltage equations, a V is displayed.
4. In the Port field, enter the number of the port that you want the equation to apply to.
5. In the Harmonic indices field, enter the harmonic index that the equation applies to, either an absolute index, or a locally-defined carrier frequency, in which case the first index must be negative.
6. In the Formula field, enter the expression that defines the current or voltage.
7. Click **Apply** to update the equation.
8. To add another equation, click **Add** and repeat steps 3-7.
9. Click **OK** to accept the changes and dismiss the dialog box.

Defining Frequency Parameters

The *freq* parameter can be used to define one or more carriers for an FDD. For more information about the freq parameter, refer to the section "Specifying Carriers with the Freq Parameter" on page 6-10.

To define a frequency parameter:

1. Double-click the FDD component to open the Edit Component dialog box.
2. Select any parameter in the Select Parameters list.
3. Click **Add**. The new parameter is automatically selected.
4. From the Parameter Entry Mode list, choose **Frequency**. The left side of the equation is changed to **Freq[n]**, where *n* is an index indicating that it is the *nth* frequency parameter defined for the FDD.
5. In the Index field, enter the index that identifies the frequency.

Note This index is used only to specify which frequency parameter to use when more than one envelope is specified for an FDD. It does not specify a frequency offset.

---

6. In the Formula field, enter the expression that defines the frequency.
7. Click **Apply** to update the parameter.
8. Click **OK** to accept the changes and dismiss the dialog box.

Defining Triggers

Up to 31 triggers can be defined for a single FDD. Any time the value of the trigger expression is equal to a value other than zero, a trigger event is declared for that trigger. Each trigger keeps a count of the number of times the trigger occurred and the time of the last trigger. For more information about triggers, refer to the section "Trigger Events" on page 6-16.

To define a trigger:

1. Double-click the FDD component to open the Edit Component dialog box.
2. Select any equation in the Select Parameters list.
3. Click **Add**. The new equation is automatically selected.
4. From the Parameter Entry Mode list, choose **Trigger**. The left side of the equation is changed to **Trig[n]**, where *n* identifies the trigger.
5. In the Index field, enter the value that identifies the trigger, 1-31.
6. In the Formula field, enter the expression that defines the trigger event.
7. Click **Apply** to update the parameter.
8. Click **OK** to accept the changes and dismiss the dialog box.

Defining Clock Enables

Clock enables restrict FDD voltages and currents to change only when a specified trigger, or a set of specified triggers, occurs. This is done by setting the clock enable of the desired port to a binary value, where the Nth bit corresponds to whether this port should be enabled by the Nth trigger. For more information, refer to the section "Output Clock Enables" on page 6-17.

To define a clock enable:

1. Double-click the FDD component to open the Edit Component dialog box.
2. Select any equation in the Select Parameters list.
3. Click **Add**. The new equation is automatically selected.
4. From the Parameter Entry Mode list, choose **Clock Enable**. The left side of the equation is changed to **ce[n]**.
5. In the Port field, enter the number of the port that you want the clock enable to apply to.
6. In the Formula field, enter the binary expression using the *bin()* function, where the Nth bit corresponds to whether this port should be enabled by the Nth trigger. For example, if you want port *n* output to be updated whenever either trigger 1 or trigger 3 occur, you would enter a value of *bin(101)* or 5 for the clock enable parameter.
7. Click **Apply** to update the parameter.
8. Click **OK** to accept the changes and dismiss the dialog box.

FDD Examples

This section offers the following examples that show how to use frequency-domain defined devices to define a variety of nonlinear circuit components. The examples include:

You can find these examples in the software under the Examples directory in this location:

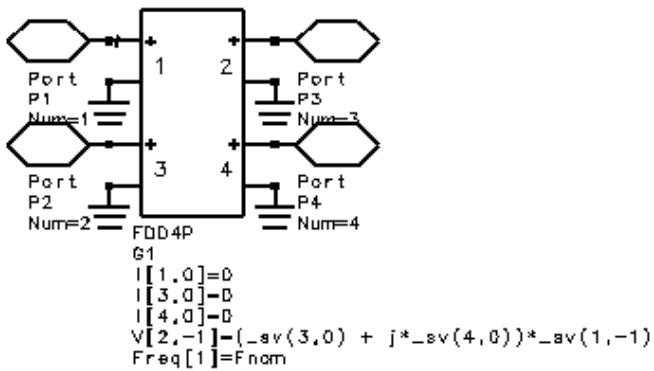
Tutorials/FDD_Examples_prj/networks

IQ Modulator

This example is under the Examples directory in the following location:

Tutorials/FDD_Examples_prj/networks/IQ_modulator.dsn

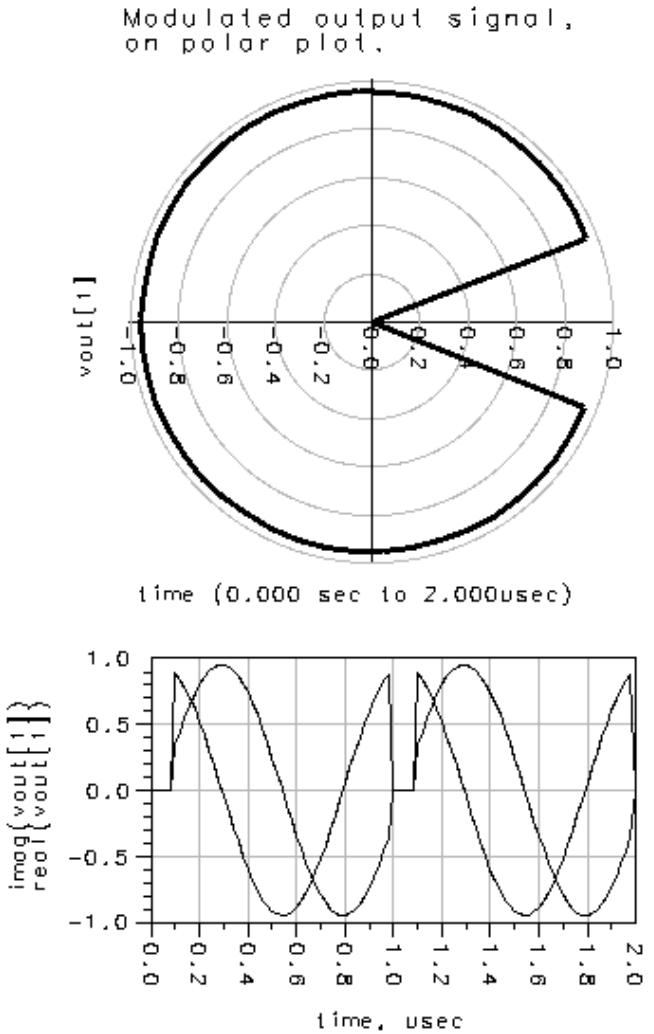
This is a simple, IQ modulator. The input signal is at port 1. The I and Q data (baseband, time-domain signals) are made available at ports 3 and 4, respectively.



Note the following points:

- I data is the baseband time-domain signal and is applied to port 3.
- Q data is the baseband time-domain data and is applied to port 4.
- The current equations for ports 1, 3, and 4 set the current at these ports to zero. These ports are treated as open circuits at all frequencies.
- The voltage equation at port 2 equates the spectral voltage at port 2 at frequency *Freq[1]* to the baseband spectral voltage at port 3 (the I signal) plus *j* times the baseband spectral voltage at port 4 (the Q signal) multiplied by the spectral voltage at port 1 at frequency *Freq[1]*.
- Note the use of -1 in the left side of the voltage equation and in the function *_sv(1, -1)*. The minus sign is required when referring to the index of a carrier defined using the *Freq* parameter, in this case, the index 1 that identifies *Freq[1]*.
- *Freq[1]* is a user defined parameter whose value is passed to the FDD.

The design *IQmodTest.dsn* shows this device under test. I and Q modulation are applied to ports 3 and 4. A 1 V, 1 GHz signal is applied to the input. The modulated output is shown here.

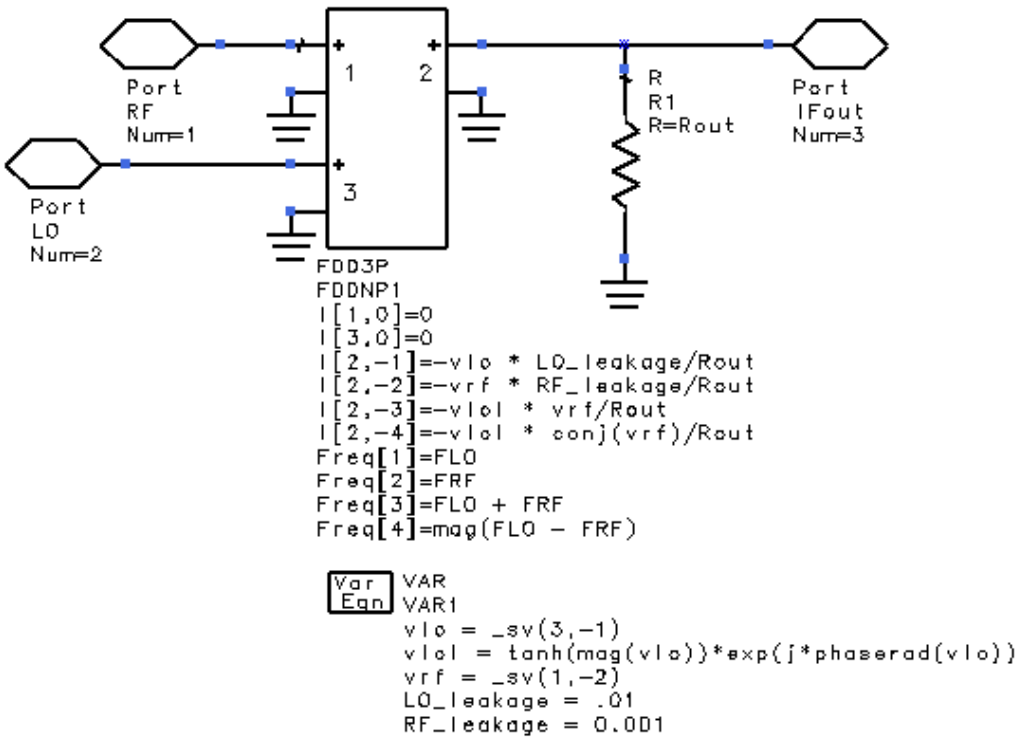


Mixer

This example is under the Examples directory in the following location:

Tutorials/FDD_Examples_prj/networks/FDDmixer.dsn

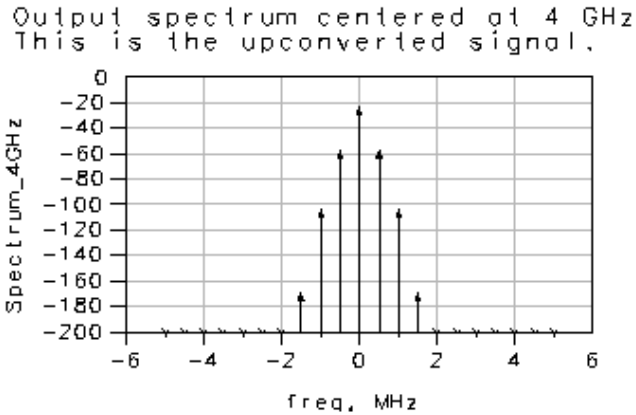
This is a simple, ideal mixer. It models upconversion, downconversion, LO leakage, RF leakage, and conversion gain compression with increasing LO amplitude



Note the following points:

- Like the IQ modulator example, ports 1 and 3 are open circuits for all frequencies.
- The signal at port 2 has four spectral components, which are defined with four current equations. The equations define:
  - LO leakage at frequency FLO
  - RF leakage at frequency FRF
  - The upconverted signal FLO+FRF
  - The downconverted signal, which is the magnitude of FLO - FRF
- Note the use of minus signs in the left side of the current equations. The minus sign is required when referring to the index of a carrier defined using the Freq parameter. In this case, *-1*, *-2*, *-3*, and *-4* each refer to *Freq[1]*, *Freq[2]*, *Freq[3]*, and *Freq[4]*, respectively.
- The use of minus signs in the right side of the current equations is necessary because the equations define positive current flowing into each port of the FDD. Thus, the minus sign changes the direction of positive current.
- The variables *Rout*, *FLO*, and *FRF* are user-defined parameters whose values are passed to the FDD.

The designs *FDDmixerTest.dsn* and *FDDmixerTestEnv.dsn* show the mixer under test in a harmonic balance simulation and Circuit Envelope simulation, respectively. One result of the Circuit Envelope simulation is shown here.


Output spectrum centered at 4 GHz
This is the upconverted signal.
Spectrum_4GHz
0
-20
-40
-60
-80
-100
-120
-140
-160
-180
-200
-6
-4
-2
0
2
4
6
freq, MHz

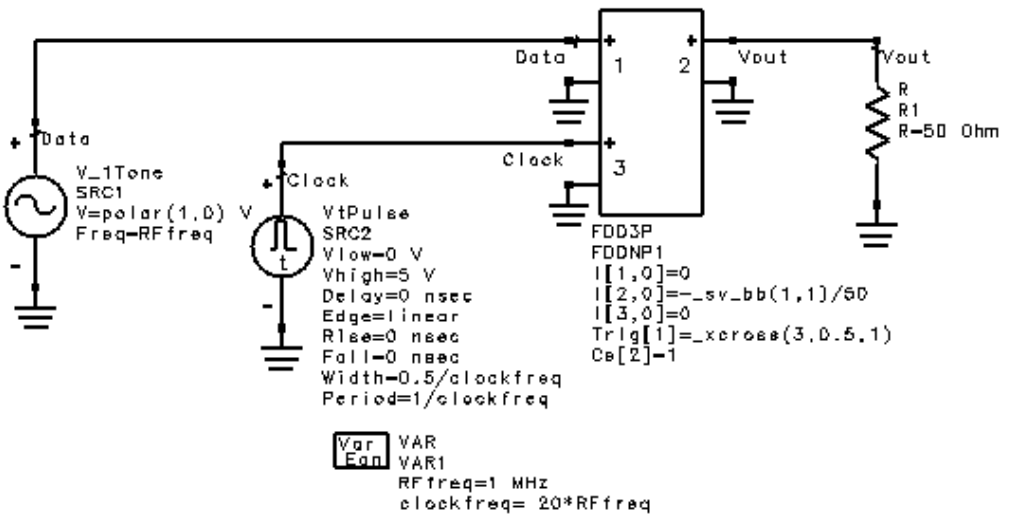
Sample and Hold

This example is under the Examples directory in the following location:

Tutorials/FDD_Examples_prj/networks/SampleHold.dsn

This is a simple sample and hold device. The FDD samples the input data, in this case, a sine wave, once per rising edge of the clock, then holds the value so that the current is constant at the output. 20 samples are take per period of the input signal.

This example also uses the trigger and clock enable features of the FDD.



Note the following points:

- Ports 1 and 3 are open circuits for all frequencies.
- The current equation for port 2 is based on the function *_sv_bb(port, trigger)*. This function, when passed a port number and trigger index, returns the total, real voltage at the port, at the last time the trigger occurred. So the current at port 2 is equal to the total, real voltage at port one at the last time trigger 1 occurred, divided by 50 ohms.
- The trigger parameter is based on the function *_xcross(port, threshold, direction)*. Given the values that are passed to this function here (3, 0.5, and 1), a trigger occurs if the baseband voltage at port 3 passes through 0.5 V, in the positive direction. For more information on triggers, refer to the section "Trigger Events" on page 6-16.
- The clock enable parameter, *Ce*, enables the output of a port to change only when a specified trigger occurs. In this instance, it means that the output at port 2 will change only when trigger 1 occurs. This produces the "hold" effect of the sample and hold device. The trigger indices must be specified in binary format. For more information, refer to the section, "Output Clock Enables" on page 6-17.

Chapter 7: User-Compiled Models Dialog Box Reference

New User-Compiled Model dialog box

Allows you to provide a filename for your new user-compiled model. You must use a name that does not already exist in the current project and is not the name of a primitive used by the simulator. Type the desired name in the Model Name box, then click OK.

One of the following two dialog boxes will appear:

- If the Analog/RF simulation type is active, the Circuit Type dialog box appears, allowing you to select a Linear or Nonlinear model before specifying parameters and other data in the three-tabbed User-Compiled Circuit Model box.
- If the Digital Signal Processing simulation type is active, the three-tabbed User-Compiled Signal Processing Model box appears.

Related Topics

User-Compiled Circuit Model, Parameters Tab

User-Compiled Circuit Model dialog box (Model Code Tab)

User-Compiled Circuit Model dialog box (Model Code Tab)

Open File dialog box

User-Compiled Circuit Model dialog box (Options tab)

Open File dialog box

Circuit Type dialog box

Allows you to specify whether your new Analog/RF user-defined model will be linear or nonlinear. Click on the appropriate choice, then choose OK.

Open User-Compiled Model dialog box

Enables you to open a previously saved user-compiled model.

Active Projects

Use the drop-down list to select the project where the user-compiled model that you want to open is stored.

Models in Project

Click the name of the model that you want to open.

OK

Click this button to open the selected model and dismiss this dialog box.

Cancel

Click this button to cancel the selected model and dismiss the dialog box.

Related Topics

User-Compiled Circuit Model, Parameters Tab

User-Compiled Circuit Model dialog box (Model Code Tab)

User-Compiled Circuit Model dialog box (Model Code Tab)

Open File dialog box

User-Compiled Circuit Model dialog box (Options tab)

Open File dialog box

User-Compiled Circuit Model, Parameters Tab

Enables you to define parameters and other specifications for your Analog/RF user-defined model.

Component Name

Displays the previously assigned name of the selected user-defined model.

Select Parameter

Lists all of the parameters defined for this model.

Add

Allows you to add a newly defined parameter to the Select Parameter list after specifications are made in the fields listed under Edit Parameter.

Cut

Removes a selected parameter from the Select Parameter list.

Paste

Inserts the last parameter that was cut from the Select Parameter list.

Add Multiplicity Factor (_M)

Add a multiplicity factor to the model.

Copy Parameters From

Choose this button to bring up the Library List dialog box to choose a component. When you select OK, the selected component's parameters will be copied to the parameter list and the associated design (.dsn) file will be copied and saved as <component_name> .dsn in the networks directory of the current project. The Layout ArtworkType and Name are also copied.

Parameter Name

Type in an appropriate parameter name.

Value Type

Allows you to select a real (default), integer, complex, or string value type.

Default Value (e.g., 1.23e-12)

Specify the value to be assigned to each parameter when the component is first inserted.

Parameter Type

(Optional) From the drop-down list, select the type of scale factors to be assigned to the entered value (for example, Ohms, KOhms, MOhms, etc. for resistance). Default is Unitless.

Parameter Description

(Optional) Type in a brief description of the parameter. Default is blank.

Display parameter on schematic

If you check this box, the parameter name and value will be displayed on the schematic.

Optimizable

If you check this box, the parameter can be optimized.

Allow statistical distribution

If you check this box, the parameter can be used in yield analysis.

OK

Choose this button when you have made all desired specifications in the tabs of the User-Compiled Circuit Model dialog box. The component information will be written to a file called <component_name>.ael and saved to the database. The program checks for a <component_name>.c file and will prompt you to generate one if such a file is not found. The dialog box is dismissed.

Apply

Choose this button when you have entered all specifications for a particular parameter.The template file will be regenerated to update the parameter data. The component information will be written to a file called <component_name>.ael and saved to the database.

Cancel

Choose this button to cancel any parameter data you have changed in the Parameters Tab. This will cancel all changes since the previous save, dismiss the User-Compiled Circuit Model dialog box. Also, if the button “Create/Edit Symbols and Pins” was selected in the Model Code tab, the Schematic window that was opened in symbol view will now close.

User-Compiled Circuit Model dialog box (Model Code Tab)

Allows you to generate, modify, and compile your Analog/RF user-defined model.

Create Edit Symbol and Pins

Opens a Schematic window in the Symbol View mode. You can create or edit a symbol. The symbol is loaded if it exists. If the <component>,dsn file is not found, the SYM_CUI.dsn file is copied to the current project's network directory as <componen>.dsn and loaded.

Code Options

Brings up the Code Options dialog box for setting specifications for the code, such as type of analysis function.

Create New Code Template

Copies a template file, $HPEESOF_DIR/modelbuilder/lib/cui_circuit.template to the current project networks directory with the name <component>.c. A warning message is displayed if a .c file exists with the same name in the current project's networks directory. If you continue, the .c file will be overwritten.

Open File

Brings up the Open file dialog box, so you can browse through your directory structure to select the pathname for the file you want to open.

Save

Saves all current edits to the active file.

Save As

Brings up the Save As dialog box so you can save all current edits to a new file with a different name.

Compilation Status

Displays the error/warning messages of the last compile and link process.

Compile Simulator/Compile Model

Generates the makefile (modelbuilder.mak), auto-generates the simulator boot procedure using the names of the object files, and executes the make process. If the selection “Recompile all out-of-object files” is checked in the Compile Options dialog box (see following description), any other .o(bj) files specified in the environment variable USER_MODEL_LINK_LIST will be recompiled if necessary, and linked into the executable. (This environment variable is located in the file de_sim.cfg in the current project directory.)

Under the path $HPEESOF_DIR/modelbuilder/lib/hpeesof, the files hpeesofdebug.mak (or hpeesofopt.mak) and user.mak are copied to the current project's networks directory if they don't exist. (Either hpeesofdebug,mak or hpeesofopt.mak is coped, depending on the selected setting of “set debug flag on” in the Compile Options dialog box.) Do not change the makefile (hpeesof debug.mak or hpeesofopt.mak). But you can edit the user.mak file. This button's label and corresponding function (compiling the simulator or only compiling the current model) are also determined by selections in the Compile Options dialog box.

Compile Options

Brings up the Compile Options dialog box, which allows you to set specifications for the compilation process.

If OK is selected, the files to be linked are saved in the environment variable USER_MODEL_LINK_LIST in the file de_sim.cfg in the current project directory.

OK

Choose this button when you have made all desired specifications in the tabs of the User-Compiled Circuit Model dialog box. The component information will be written to a file called <component_name>.ael and saved to the database. The program checks for a <component_name>.c file and will prompt you to generate one if such a file is not found. The dialog box is dismissed.

Apply

Choose this button when you have entered all specifications for a particular field in the Model Code tab. The specified data will be saved to the appropriate files and the dialog box will remain open.

Cancel

Choose this button to cancel any data you have specified in the User-Compiled Circuit Model box. This will cancel all changes, dismiss the User-Compiled Circuit Model dialog box, and close the Schematic window (if it is open).

Related Topics

Code Options (Analog/RF Models) dialog box

Compile Options dialog box

Open File dialog box

Allows you to browse through your directory structure to select the pathname for the code file that you want to open.

Filter box

Displays the currently selected directory.

Directories

List of accessible directories.

Files

List of user-defined code files in the currently selected directory

Selection

Type in a pathname for the directory where your user-defined code is stored or change directory paths in the Directories box until the correct path is shown here.

OK

Opens the selected code file and dismisses the dialog box.

Filter button

Change to the selected directory and display all user-defined code files from this directory in the Files box.

Cancel

Cancel the selected pathname and dismiss the dialog box.

Save As dialog box

Allows you to browse through your directory structure to select the pathname for the code file that you want to open.

Filter box

Displays the currently selected directory.

Directories

List of accessible directories.

Files

List of user-defined code files in the currently selected directory

Selection

Type in a pathname for the directory where you want to save the user-defined code file followed by the desired filename or change directory paths in the Directories box until the correct path is shown here, then type in the filename at the end of the path.

OK

Creates the selected code file as specified and dismiss the dialog box.

Filter button

Change to the selected directory.

Cancel

Cancel the selected pathname and dismiss the dialog box.

Code Options (Analog/RF Models) dialog box

Allows you to specify certain specifications for the code, such as type of analysis function. Check the boxes for the items that you want to include. By default, no functions are checked. These values are not saved in the <component_h>.c file until you choose the Create New Code Template button.

Compile Options dialog box

Allows you to specify the option files in a particular project that are to be linked to or unlinked from the simulator executable.

Active Projects

Use the drop-down list to select the appropriate project from the list of available projects.

Objects Files in Project

Select the object files that you want to link to the project, then click the right-arrow button.

Object Files to Be Linked

Includes the previously compiled model files for the project shown. Select the object files that you want to unlink from the project, then click the left-arrow button.

If compilation is successful, make executable

If you check this box, an executable file for the user-defined model will be created upon completion of a successful compilation. Default is checked. Note that if a previously built simulator executable is running, you must first stop and release it.

Recompile all out-of-date object files

If you check this box, any other .o(bj) files specified in the USER_MODEL_LINK_LIST will be recompiled if necessary, and linked into the executable.Default is not checked.

Set debug flag on

If you check this box, the debug flag is enabled.Default is checked.

OK

Click to dismiss this dialog box. All object files in the box on the right of this box will be linked.

Cancel

Cancels all specified data in this box and dismiss the dialog box only.

User-Compiled Circuit Model dialog box (Options tab)

Allows you to specify a variety of options associated with the user-defined model.

Component Description

Type in a descriptive name for the component. The default, as shown in the box, is User-Compiled Model. This will be the identifier used for this component when it is edited in the Edit Component dialog box, which is used for modifying component parameters in the Schematic window.

Component Instance Name

Optional.The name specified in this box is used as a prefix in building a unique name (ID) for every user-defined item. It is part of the annotation displayed with the symbol representing the parameter subnetwork when placed in a design. The default is the name of the component.

Library Name.

Optional. The name of the library where the component is located. The default is My Circuit Library. Type in a new name if desired.

Layout Artwork (Type)

Optional. You can specify an artwork representation to be associated with the component in the Layout window. The available types are Synchronized, Fixed, AEL Macro, and None. None is the default.

Layout Artwork (Name)

Optional. You can select the name of an artwork representation to be associated with the component in the Layout window if you select Fixed or AEL Macro in the Layout Artwork (Type) field.

Include in Bill of Materials

Optional. if this box is checked, the instances of the user-defined component will be logged in the Bill of Materials. Default is not checked.

Use External Text Editor

If this box is checked, a text editor will start. The default editor used is the hpeesofedit editor. If you wish to access a different text editor, type in its pathname, then select the check-box. (The vi editor is not available.)

OK

Choose this button when you have made all desired specifications in the tabs of the User-Compiled Circuit Model dialog box. The component information will be written to a file called <component_name>.ael and saved to the database. The program checks for a <component>.c file and will prompt you to generate one if such a file is not found. The dialog box is dismissed.

Apply

Choose this button when you have entered all specifications in the Option tab. The specified data will be saved to the appropriate files and the dialog box will remain open.

Cancel

Choose this button to cancel any data you have specified in the User-Compiled Circuit Model dialog box. This will cancel all changes, dismiss the User-Compiled Circuit Model dialog box, and close the Schematic window (if it is open).

Delete User-Compiled Model dialog box

Allows you to delete a previously saved user-defined model. From the Active Projects field, use the drop-down list to select the appropriate project from the list of available projects. From the Models in Project box, click the name of the model(s) that you want to delete. Click Cancel to dismiss the dialog box only. Click Apply to delete the selected model(s) and leave the dialog box open. Click OK to delete the selected model(s) and dismiss the dialog box.

Link User-Compiled Model dialog box

Allows you to link or unlink selected object files to the simulator. The Link Status box displays error/warning messages from the previous link. To dismiss the dialog box only, select Cancel. To add or delete options to be linked, choose the Link Options button, which brings up the Link Options dialog box. Make specifications in the Compile Options dialog box, then choose the Link Simulator button to link all selected object files.

Related Topics

Compile Options dialog box

Link Options dialog box

Allows you to specify the object files from a particular project to be linked or unlinked. From the Active Projects field, select an active project from the drop-down list. The large box on the left shows object files that exist in the selected project. The large box on the right specifies object files to be linked. To add to the list of files to be linked, select the appropriate file to be linked from the left box, then click the right-arrow button. To remove files from the list to be linked, select the appropriate file from the right box, then click the left-arrow button. If you wish to cancel changes that you have entered and dismiss the box, choose Cancel. If you want to activate the changes you have made in the box, choose OK.

Confirm Model File Not Created message

This message appears when you attempt to dismiss this dialog box without having compiled a model. Click Yes if you wish to compile a model at this point. Otherwise, click No.

Confirm Files Out-of-Synch message

This message occurs when you attempt to dismiss the active dialog box without having linked recently created files that are more current than the existing executable. If you wish to dismiss the box without linking the new models, click yes. Otherwise, click No.

File/Directory Management dialog box, UNIX

The fields described below are found in a number of dialog boxes used in various file and directory management operations.

Filter

Displays the current path and indicates which type of files will be listed in the Files list box. For example, the Delete Design File dialog box Filter field is set to *.dsn in the networks subdirectory to display all design files there.

Directories

Displays the directories on the current file system. Double-click to traverse the directory structure until the Files list box displays the file or directory you are looking for.

Files

Displays all files or directories, meeting the Filter criteria above, in the selected directory. Double-click the file or directory you want, or click once and choose OK.

Selection

Reflects the currently selected path and filename. This field should reflect the file or directory you want to open before you choose OK.

Tips:

- If you change the path by typing in the Filter field, click the Filter button below to update the Files list box.
- You can traverse the Directories list box with a single click (rather than a double-click) if you click the Filter button each time you select a directory.
- In the Directories list box, double-click the line with the double dots to move up one level.

File Management dialog box, PC

The fields described below are found in a number of dialog boxes used in various file management operations.

Look in:

Displays the current directory (folder). Select a different directory from the drop-down lists or double-click a folder displayed in the box below. Click the folder icon with an arrow to view the folders up one level.

File name:

Displays the selected file.

Files of type:

Displays the file extension used to filter the files in the current directory.

Tip:

On NT 4.0, click the question mark, position the pointer over the dialog box control you want information on, and click. The help provided by Windows for that control is displayed.

Directory Management dialog box, PC

The fields described below are found in a number of dialog boxes used in various directory management operations.

Directories

Displays the contents of the current directory. Double-click as needed to locate desired directory. Double-click to select it. The new path is reflected under the “Directories:” label.

Drives

Displays the current drive. If necessary, use the drop-down list to select a different one.

Network

If necessary, click Network to display the Connect Network Drive dialog box for connecting to a different drive on the network.

Index