cadence®

Component Description Format User Guide

Product Version 6.1.4

February 2011

Contents

Preface

The Component Description Format (CDF) describes the parameters and the attributes of parameters of individual components and libraries of components. The CDF lets you create and describe your own components.

This user guide describes how you can use the Component Description Format to create and describe your own components. This manual assumes that you are a computer-aided design librarian or a circuit designer and that you are familiar with designing and developing electronic components with Virtuoso® design and simulation software.

This preface describes the following:

Related Documents for Component Description Format

Component Description Format (CDF) descriptions are often used with other Cadence products during the design process. The following give you more information about the use of CDF descripti ons.

- *Virtuoso Parameterized Cell Reference* describes using CDF parameters with *parameterized cells*.
- *Virtuoso Schematic Editor L User Guide* describes the Edit Component Display Options form and label specifications.
- *Analog Expression Language Reference* describes the analog expression language syntax frequently used in CDF descriptions.

Third Party Tools

To view any `.swf` multimedia files, you need:

- A SourceLink Login.
- Flash-enabled web browser, for example, Internet Explorer 5.0 or later, Netscape 6.0 or later, or Mozilla Firefox 1.6 or later. Alternatively, you can download Flash Player (version 6.0 or later) directly from the Adobe website.
- Speakers and a sound card installed on your computer for videos with audio.

Typographic and Syntax Conventions

This book uses two kinds of conventions, one to help you use menu commands and the other to help you use Cadence SKILL language functions.

Typographic Conventions

Boxes and arrows in a sequence like the ones below show you the order in which you select a command from a menu.

Tools ➢ **CDF** ➢ **Copy ...**

Syntax Conventions

This list describes the syntax conventions used for the SKILL functions specified in this manual.

`literal (LITERAL)`	Nonitalic (UPPERCASE) words indicate keywords that you must enter literally. These keywords represent command (function, routine) or option names.
`argument (z_argument)`	Words in italics indicate user-defined arguments for which you must substitute a name or a value. (The characters before the underscore (_) in the word indicate the data types that this argument can take. Names are case sensitive. Do not type the underscore (*z_*) before your arguments.)
`[ ]`	Brackets denote optional arguments.

...	Three dots (...) indicate that you can repeat the previous argument. If you use them with brackets, you can specify zero or more arguments. If they are used without brackets, you must specify at least one argument, but you can specify more.
	`argument...: specify at least one, but more are possible`
	`[argument]...: you can specify zero or more`

,...	A comma and three dots together indicate that if you specify more than one argument, you must separate those arguments by commas.
`{ }`	Used with vertical bars and encloses a list of choices from which you must choose one.
`|`	Separates a choice of options.

If a command line or SKILL expression is too long to fit inside the paragraph margins of this document, the remainder of the expression is put on the next line, indented. For example:

```
?callback "cdfgData->pName1->value = cdfgData->pName2->value")
```

When writing the code, you should always put a backslash (\) at the end of any line that continues on to the next line.

Identifiers Used to Denote Data Types

The Cadence SKILL language supports several data types to identify the type of value you can assign to an argument.

Data types are identified by a single letter followed by an underscore; for example, `t` is the data type in `t_viewNames`. Data types and the underscore are used as identifiers only; they should not be typed.

Prefix	Internal Name	Data Type
`a`	array	array
`b`	ddUserType	Boolean
`C`	opfcontext	OPF context
`d`	dbobject	Cadence database object (CDBA)
`e`	envobj	environment
`f`	flonum	floating-point number
`F`	opffile	OPF file ID
`g`	general	any data type
`G`	gdmSpecIlUserType	gdm spec
`h`	hdbobject	hierarchical database configuration object
`l`	list	linked list
`m`	nmpIlUserType	nmpIl user type
`M`	cdsEvalObject	—
`n`	number	integer or floating-point number
`o`	userType	user-defined type (other)
`p`	port	I/O port
`q`	gdmspecListIlUserType	gdm spec list
`r`	defstruct	defstruct
`R`	rodObj	relative object design (ROD) object
`s`	symbol	symbol
`S`	stringSymbol	symbol or character string

Prefix	Internal Name	Data Type
t	string	character string (text)
u	function	function object, either the name of a function (symbol) or a lambda function body (list)
U	funobj	function object
v	hdbpath	—
w	wtype	window type
x	integer	integer number
y	binary	binary function
&	pointer	pointer type

For information on the SKILL language, see the *Cadence SKILL Language User Guide*.

1

What is CDF?

Overview

The Component Description Format (CDF) describes the parameters and the attributes of parameters of individual components and libraries of components. The CDF lets you create and describe your own components. A CDF description assigns parameters and parameter attributes to libraries and cells for many purposes:

- Assigning parameter names and values
- Allocating units and default values
- Checking that values lie within specified ranges
- Dynamically changing how parameters are displayed depending on predefined conditions
- Executing a SKILL callback function whenever certain information is changed

  Note: Callbacks are called only from the Add Instance and Edit Object Properties forms. As a consequence, many problems are associated with callbacks and their use is discouraged.

The CDF gives you:

- a single location for storing all component information so that applications can share and reuse parts of the component description.
- independence from applications and cellviews.
- a graphical user interface (the Edit CDF form) for entering and editing component information.
- strict checking so that tools do not need to check component data before using it.
- basic units and scale factors that are common in electrical design.
- a modular format, so that you can edit and update parts of a description.
- a command to copy the CDF description of one component into another component.

A Typical Application

The CDF encompasses all levels of components, including discrete and full-custom components, and all cellviews of components. For example, the resistor cell *res* can take on two forms: an ideal resistor or a thin film resistor. If you want an ideal resistor, you specify the value of resistance desired. If you want a thin-film resistor, which models discontinuities as

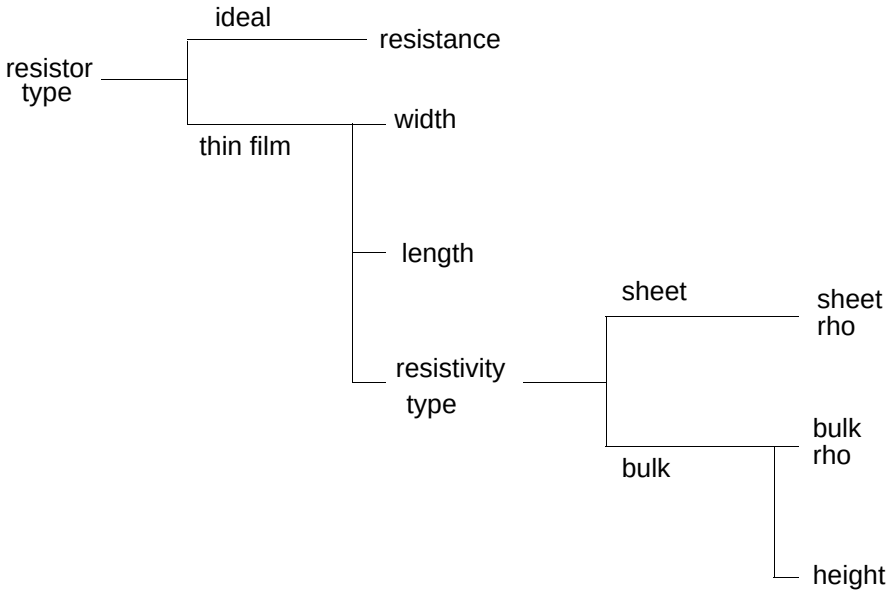
well as skin and substrate effects, you specify the length and width of the resistor and the resistivity of the material in which the resistor body is built.

You can choose how to specify the thin film resistivity by

- Specifying the resistivity as a sheet resistance (ohms per square)
- Specifying the bulk resistivity of the material

  In this case, you need to also specify the height of the resistor body.

To determine what parameters the resistor needs, follow the decision tree in the following figure:



This pattern of resistor parameters illustrates how you can construct a CDF description for this resistor that lets you automatically switch between different choices.

To create a complete CDF for this resistor, you create eight parameters.

The first parameter is the radio button field, *resistorType*. It is always displayed and editable and has only two values, *ideal* and *thin_film*.

The second parameter, *resistance*, uses a *cdfgData* variable expression to test the value of *resistorType*. If *resistorType = ideal*, then *resistance* is displayed and editable. If *resistorType = thin_film*, then *resistance* is not available. *resistance* has a floating-point numerical value.

The third, fourth, and fifth parameters behave in the opposite manner from *resistance*. Through another cdfgData expression for each parameter, *width*, *length*, and *resistType* become active if *resistorType* = *thin_film,* and they become unavailable if *resistorType* = *ideal*. The *width* and *length* parameters have floating-point numerical values. The *resistType* parameter is a string with only two values, *sheet* and *bulk*.

The last three parameters, *srho*, *brho*, and *height*, have floating-point numerical values. *srho* becomes active when *resistType* = *sheet*, while *brho* and *height* become active when *resistType* = *bulk*.

Additionally, you can use CDF to display prompts that spell out the complete name of each variable (*bulk rho* instead of *brho*), and you can also set the CDF of this resistor to accept expressions (such as those in the Analog Expression Language (AEL) for the values of these parameters.

When a designer accesses this resistor through the *Add Instance, Create Instance,* or *Edit Properties* command, the form prompts the designer for the necessary values. The form changes based on the values entered.

The CDF description of the resistance parameters for this resistor is stored in the following SKILL code.

```
libId = ddGetObj("base")
cellId = ddGetObj(libId "res")
cdfDataId = cdfCreateBaseCellCDF(cellId)
cdfParamId = cdfCreateParam(cdfDataId
?name           "resistorType"
?type           "radio"
?prompt         "Type:"
?defValue       "ideal"
?choices        list("ideal" "thin film")
?callback       "myModelTypeCB()")
cdfParamId = cdfCreateParam(cdfDataId
?name           "resistance"
?type           "float"
?prompt         "Resistance:"
?defValue       100.0
?callback       "myResCB()")
cdfParamId = cdfCreateParam(cdfDataId
?name           "width"
?type           "float"
?prompt         "Width:"
?defValue       20e-6
?callback       "myWidthCB()")
cdfParamId = cdfCreateParam(cdfDataId
?name           "length"
?type           "float"
?prompt         "Length:"
?defValue       100e-6
?callback       "myLengthCB()")
```

```
cdfParamId = cdfCreateParam(cdfDataId
?name           "resistType"
?type           "radio"
?prompt         "Resistivity Type:"
?defValue       "sheet"
?choices        list("sheet" "bulk")
?use            "cdfgData->modelType->value ==\"thin film\""
?callback       "myResTypeCB()")

cdfParamId = cdfCreateParam(cdfDataId
?name           "srho"
?type           "float"
?prompt         "Sheet Rho:"
?defValue       25.0
?use            "cdfgData->modelType->value == \"thin
film\"   && cdfgData->resType->value == \"sheet\""
?callback       "mySheetRhoCB()")

cdfParamId = cdfCreateParam(cdfDataId
?name           "brho"
?type           "float"
?prompt         "Bulk Rho:"
?defValue       25e-10
?use            "cdfgData->modelType->value == \"thin
film\"   && cdfgData->resType->value == \"bulk\""
?callback       "myBulkRhoCB()")

cdfParamId = cdfCreateParam(cdfDataId
?name           "height"
?type           "float"
?prompt         "Height:"
?defValue       25e-10
?use            "cdfgData->modelType->value == \"thin
film\"   && cdfgData->resType->value == dq\"bulk\""
?callback       "myHeightCB()")
```

Levels of CDF Data

You can attach a CDF description to either a component or a library of components. When you attach a description to a library, all components in the library inherit the description. When you attach a description to a component, you actually attach the description to the *cell.*

Librarians usually create and maintain the CDF descriptions for components. Library users see the effects of the descriptions as they use components from the library. However, you might want to override parts of the default description. For example, you might want to specify your own default values for CDF parameters. So that you can override the description, Cadence defines the following description levels:

- The *base-level* CDF is the description that is permanently attached to a cell or library. When the system reads in a cell or library, any base-level CDF attached to it is also read. The creator of a base-level CDF must have write permission on the object to which the description is being attached.

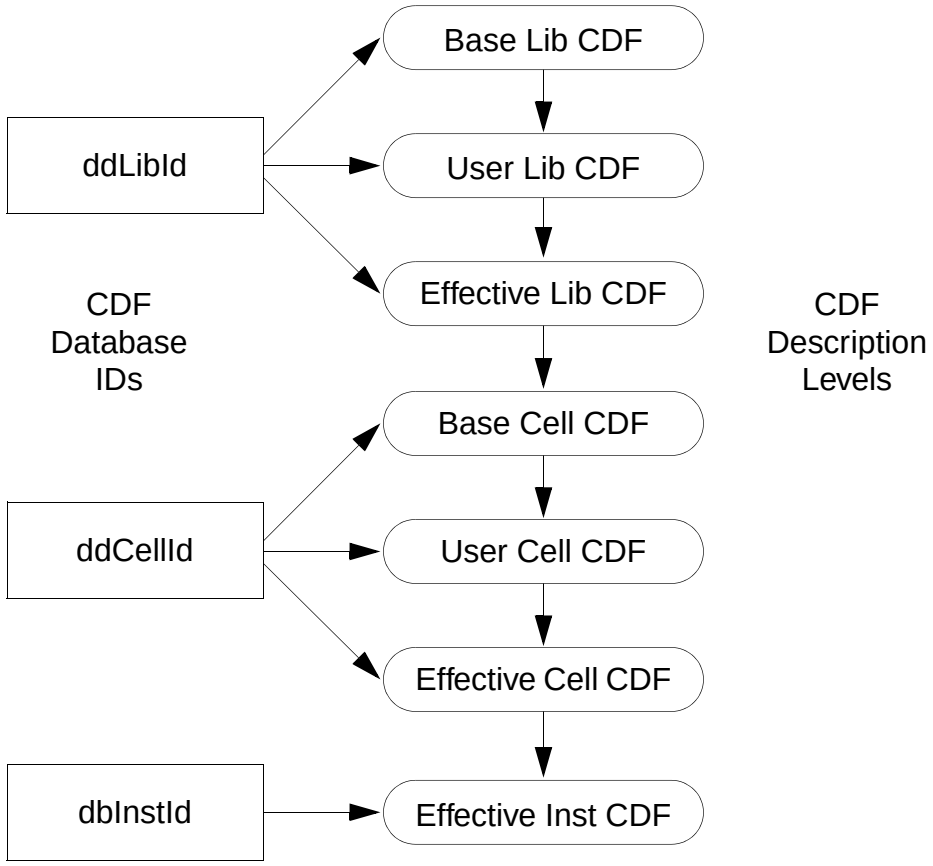
- The *user-level* CDF is a user-specific overlay description on top of the base-level CDF for a cell or a library (although the base-level does not have to exist). The system never saves a user-level CDF, so you must save it to a file and reload this level on startup. You do not need write permission to attach user-level CDF information to an object.
- The *effective-level* CDF is an overlay of a user-level CDF description on top of a corresponding base-level CDF description.

Note: Because user-level CDF is specific to a user and not a library or cell, simulation results can be different for different users. The use of user-level CDF is therefore discouraged.

“Overlay” means “effectively replaces.” A user cell value of 10,000 ohms for the resistor parameter *r* takes the place of any base- or library-level value for *r*. Each level of CDF is optional. You can create cell-level CDFs without having library-level CDFs, and you can create user-level CDFs without having base-level CDFs.

Both libraries and cells can have base- and user-level CDF descriptions. The effective-library CDF is the overlay of the user-library CDF on top of the base-library CDF. The effective cell-level CDF is the overlay of the user cell CDF on top of the base cell CDF, overlaid on top of the effective library-level CDF. Finally, the effective instance CDF is the effective cell-level

CDF with instance-specific information added to the CDF description. The following figure shows the relationship between CDF levels and the Cadence database.



CDF Worksheet

Before you start to build a component, you need to decide how you intend to use it, in which applications, and with what requirements. Before you create the cell and CDF, you need to determine the following:

1. What applications will you use?
   - ❑ Design Entry
   - ❑ Simulation
   - ❑ Layout
   - ❑ Design Checking

- ❑ Place and Route

2. What views are required by the applications you will use?

- ❑ Design Entry — symbol
- ❑ Layout — layout, flat, or *pcell*
- ❑ Simulation — *ams*, *spectre* etc.
- ❑ Design Checking — *auLvs*, *lvs*

3. What parameters are required by the applications?

- ❑ Layout — *pcell* parameters, view names
- ❑ Simulation — model parameters

4. What restrictions do the applications impose on the parameters?

- ❑ Simulation — parameter representation
- ❑ Layout — *pcell* parameters representation, correspondence between the *pcell* and CDF default values
- ❑ Are parameter values discrete values or any value in a range?

5. What dependencies do the parameters have on each other?

6. How do you want the user interface (the Add Instance and Edit Property forms) to appear?

- ❑ Are there parameters that you want to hide from the user?
- ❑ Are there parameters that you want the user to see but not be able to edit?
- ❑ What parameters should appear at the top of the forms?

This user guide illustrates each task with examples based on two components. In the first example, you examine the CDF of the functional block description of a complex pole. This component uses a macro model and is set up for simulation using the Virtuoso Spectre simulator. This example illustrates parameter passing in hierarchical designs.

In the second example, you create the CDF for a BICMOS NFET from the CDF information for another component. The NFET is specified as a parameterized cell (pcell) that can be simulated with Analog LVS. This example illustrates some of the consequences of specifying particular values for a component CDF description.

For example, the answers to the set of questions for the BICMOS NFET cell would be as follows:

1. What applications will you use?
   - ❑ Design Entry — Yes
   - ❑ Simulation — Yes
   - ❑ Layout — Yes
   - ❑ Design Checking — Yes
   - ❑ Place and Route — No
2. What views are required by the applications?
   - ❑ Schematics — symbol with G, S, and D pins for the gate, source, and drain. You do not need a bulk node connection in this case.
   - ❑ Simulation — *spectre* interfaces to the Spectre simulator.
   - ❑ Layout — *pcell* for the usual case, but also a large power transistor for I/O in a flat layout. Both layouts have G, S, and D connections.
   - ❑ Design Checking — *auLvs*
3. What parameters are required by the applications?
   - ❑ Schematics — none
   - ❑ Virtuoso XL Layout Editor (Virtuoso XL) — width, length, bend, contact information, and *maskLayoutViewName* (because there are two different layouts)
   - ❑ Spectre — model parameters, including w and l and a bulk node
   - ❑ LVS — w and l
4. What restrictions do the applications impose on the parameters?
   - ❑ Simulation — simulator parameters must be strings.
   - ❑ Layout — *pcell* parameters need to be floating-point numbers and the defaults between the CDF and the *pcell* must match.
   - ❑ The length parameter has a minimum and a maximum. The width parameter has a minimum and a maximum. The length and width must be on grid. There are no other specific discrete values.
5. Do the parameters have dependencies on each other?
   - ❑ The number of gates is a function of width. If the width exceeds 120, the layout changes to have two gates (a bend with connections).

- If a designer selects the alternate large layout, the width and length are updated to reflect the value.
- Length and width of each two parameters: one a string for simulation and one a floating-point number for the *pcell*. You should not have to enter both and they should be synchronized.
- The model parameters are dependent on the physical parameters. For example, you might calculate a value for the drain diffusion area and a value for the source diffusion area based on the physical parameters.

6. How do you want the user interface to appear? Are there parameters that you want to be hidden from designers? Are there parameters that you want the designers to see but not edit? What parameters should appear at the top of the list?

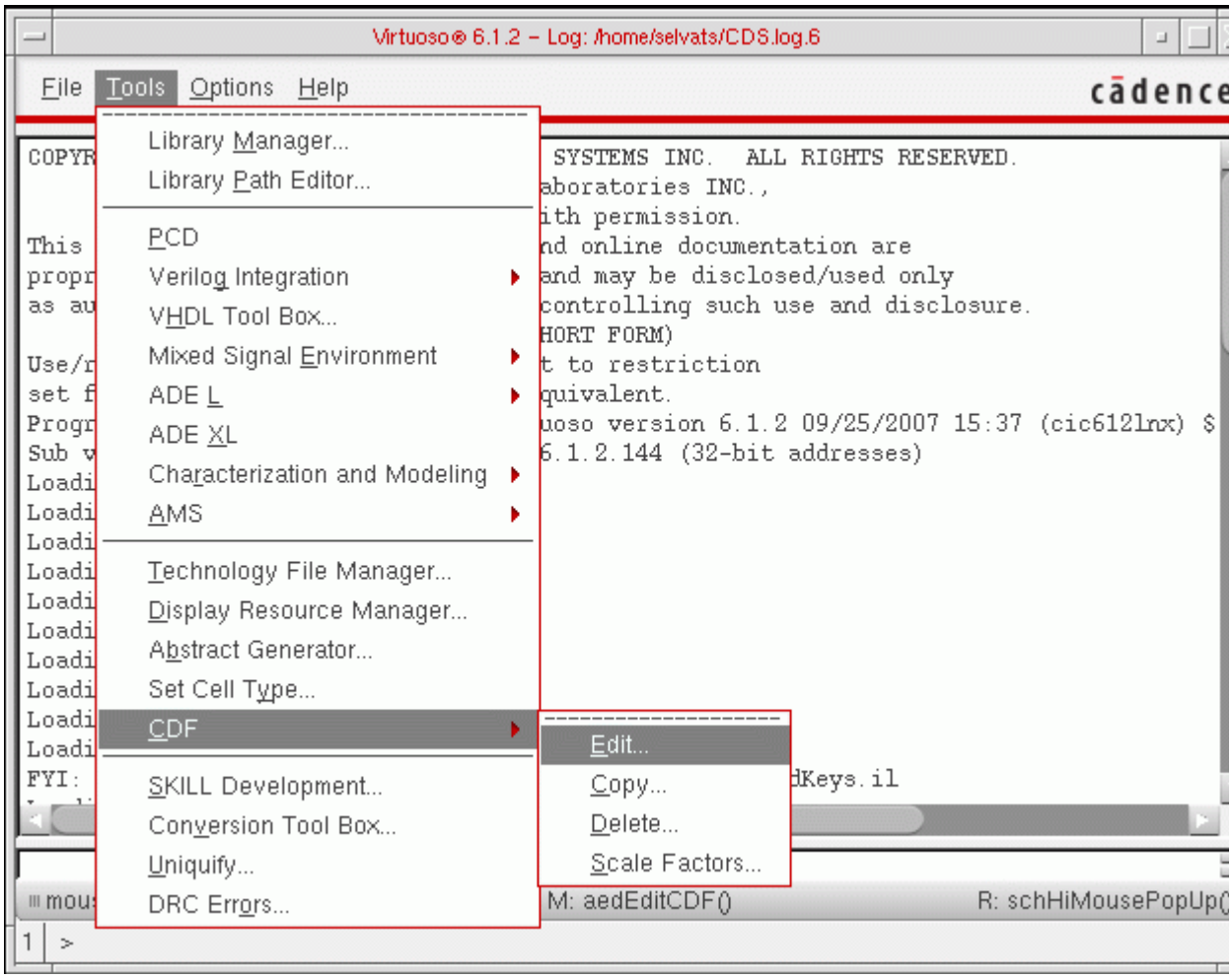
- If a designer selects the alternate large size layout, the width and length cannot be edited because there is only one large size device.
- The bends parameter is not visible to a designer because it is not under the designer's control.
- The *pcell* length and width are not visible to a designer. You want the designer to specify the width and length only once so that the forms show only the string version.
- Because drain and source diffusion areas are calculated, they should not be editable.
- Show width and length and the layout choice at the top of the forms because those are the parameters that designers must frequently edit.

2

CDF Commands

CIW CDF Commands

This section describes commands for managing Component Description Format (CDF) information. These CDF commands are on the *Tools* menu of the Command Interpreter Window (CIW). The following menu is just one of many different *Tools* menus you might see, depending on which Cadence® software you own. However, all CIW *Tools* menus have the same CDF commands.



Edit lets you modify CDF information for a cell or library.

Copy lets you copy CDF information from one cell or library to another.

Delete lets you remove CDF information from a cell or library.

Scale Factors lets you set scaling factors for displaying CDF parameters.

Edit

Tools > **CDF** > **Edit ...**

Creates or edits CDF data for a library or cell. With this command, you can

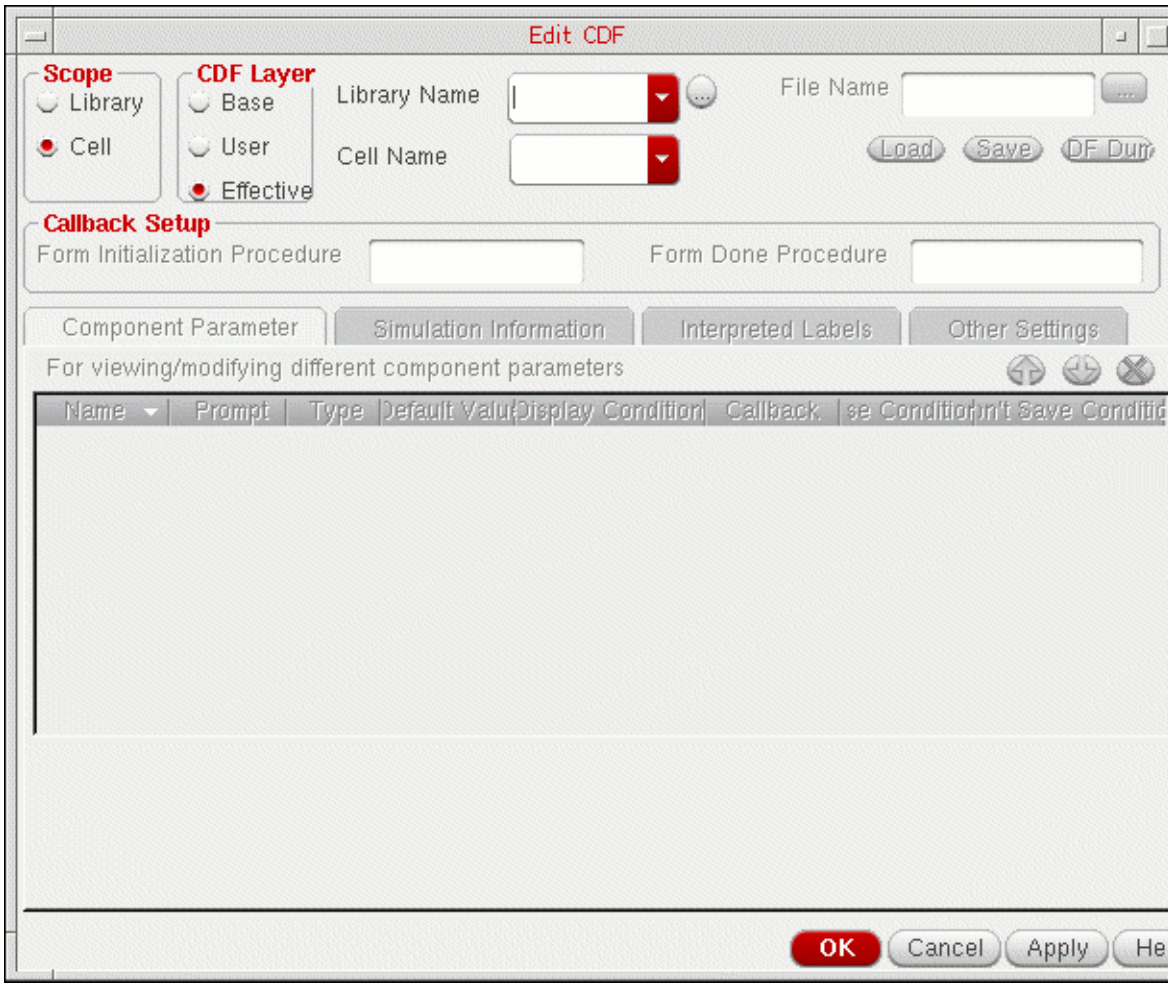
- Add component parameters
- Delete component parameters
- Modify attributes of component parameters
- Rearrange the display order of component parameters
- Modify simulation information for all available simulators
- Modify interpreted label information for schematic annotation
- Modify special CDF properties that define CDF-generated forms, such as the Add Instance form

Video

For more information, see CDF Editor Demo.

You can select CDF data for a cell or library on the Edit CDF form. Once you make a CDF selection, this form modifies itself and expands to include any existing CDF information.

Edit CDF Form (Initial)



- **Scope** lets you choose a CDF scope of *Library* or *Cell*.
  - **Library** lets you edit CDF data for a library.
  - **Cell** lets you edit CDF data for a cell. (The default value.)
- **CDF Layer** lets you choose a CDF level of *Base*, *User*, or *Effective*.
  - **Base** mode writes information to the library or cell when you click *Apply*, so you must have write privileges to the library and cell.
  - **User** mode writes data to the user-level CDF when you click *Apply*. The editor does not store data permanently in the library or cell, so you must save it to a file and reload this level on startup if you want to reuse the data. You do not need write privileges to the library or cell to attach user-level CDF information.

Note: Because user-level CDF is specific to a user and not a library or cell, simulation results can be different for different users. The use of user-level CDF is therefore discouraged.

- **Effective** mode, in contrast to *user* mode, writes information to the user-level CDF only when the new information differs from base-level CDF information. The editor does not store data permanently in the library or cell, so you do not need write privileges.

- **Library Name** is the name of the library whose CDF data you want to edit.

  Select the library name using the *Library Name* cyclic field, or click the browse button to use the Library Browser form to select the library.

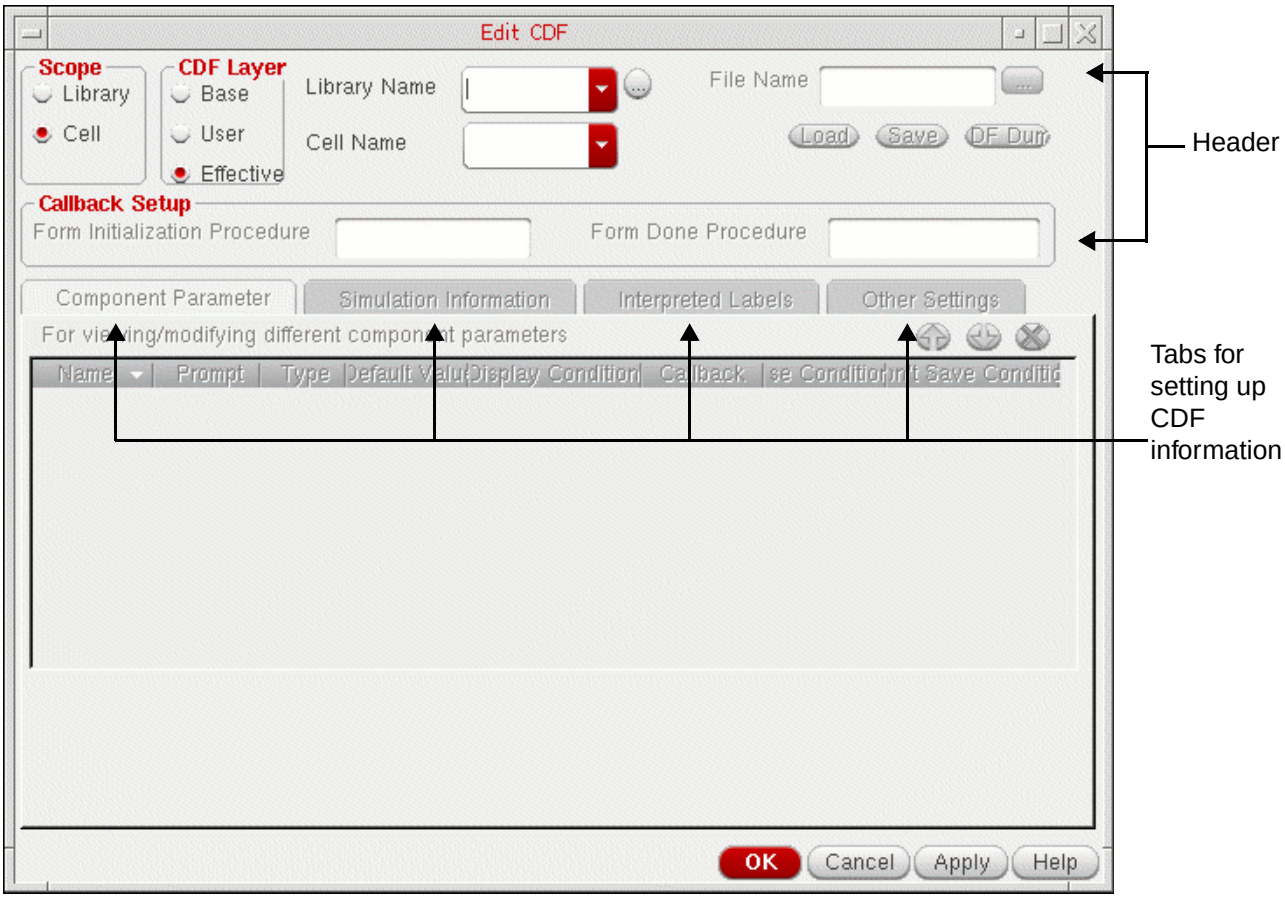
- **Cell Name** is the name of the cell whose CDF data you want to edit.

Note: If you press *Enter* or click *OK*, the values you have entered take effect, but the Edit CDF form also closes. Move the cursor or press the *Tab* key when you finish an entry in a field and want to move to another field. Click *Apply* if you want the information you entered to take effect and you want the form to stay open.

Edit CDF Form

You can use the Edit CDF form to create, view, or edit a CDF description. The Edit CDF form gives you access to all portions of cell and library CDF descriptions. It consists of a header for identifying the library or cell CDF description, specifying procedures for preprocessing and postprocessing CDF data, and four tabs for details about component parameters, simulation information, interpreted labels, and other settings.

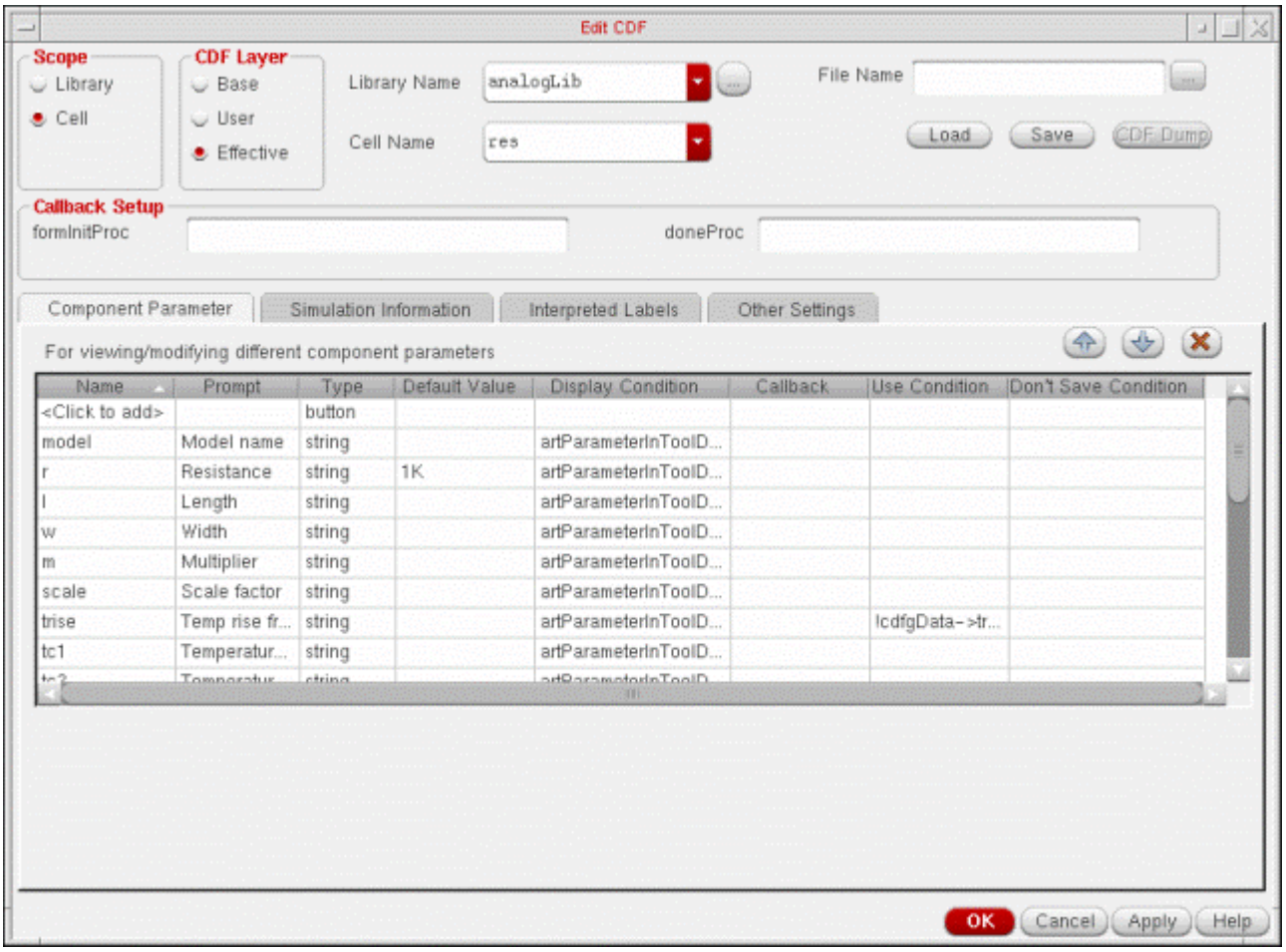
Edit CDF Form



The following sections in this chapter describe the fields and buttons in the header. Subsequent chapters describe each of the four tabs.

Edit CDF Form (Redisplayed)

After you enter a CDF cell or library name, the Edit CDF form displays component parameter information in the Component Parameter tab as shown in the following figure. Click on the other tabs to examine their contents.



File Name is the name of a file that you can create where you can store all the current editor field values. You can store the CDF description of a cell, including *User* or *Effective* levels, for future use by the Edit CDF form. You can type in the name of the file.

Load loads the CDF information in the file in the *File Name* field. The Edit CDF form redisplays to show the new information. The editor does not save CDF data to the cell or library until you click *OK* or *Apply*.

Save saves the contents of the Edit CDF form to the file in the *File Name* field.

CDF Dump saves the CDF information in SKILL format to the file in the *File Name* field. You can use the SKILL file to load the CDF information later with the `load("filename")`

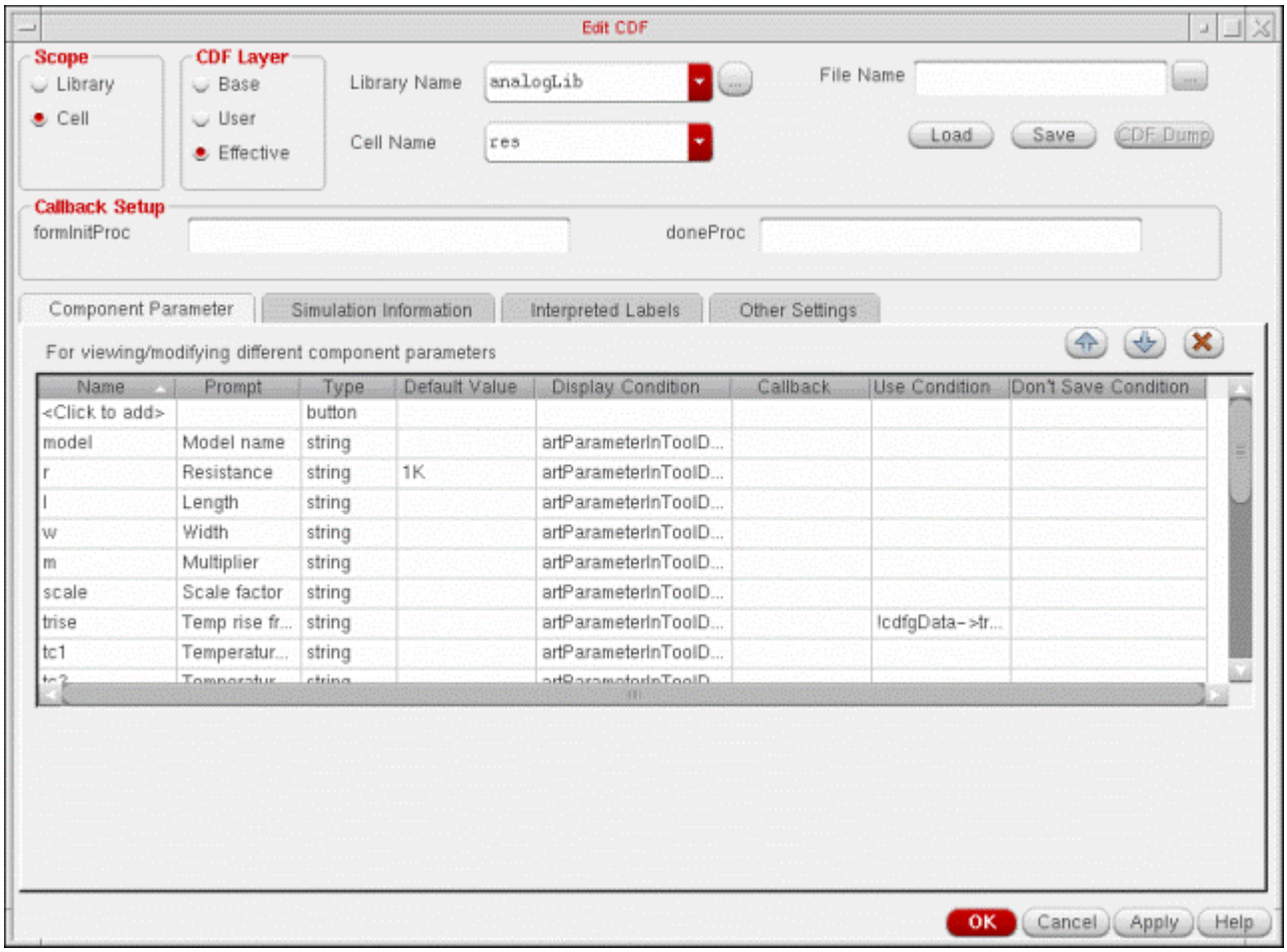
command in the Command Interpreter Window (CIW), or customize it to modify the CDF information.

formInitProc lets you specify an optional procedure (a Cadence® SKILL language routine that you provide) that executes automatically when the component is placed on an instantiation form. For more information, see Initialization Procedure on page 133.

doneProc lets you specify an optional procedure (a SKILL routine that you provide) that executes after you change any parameter on the instantiation form. For more information, see Postprocessing Procedure on page 134.

Component Parameter tab

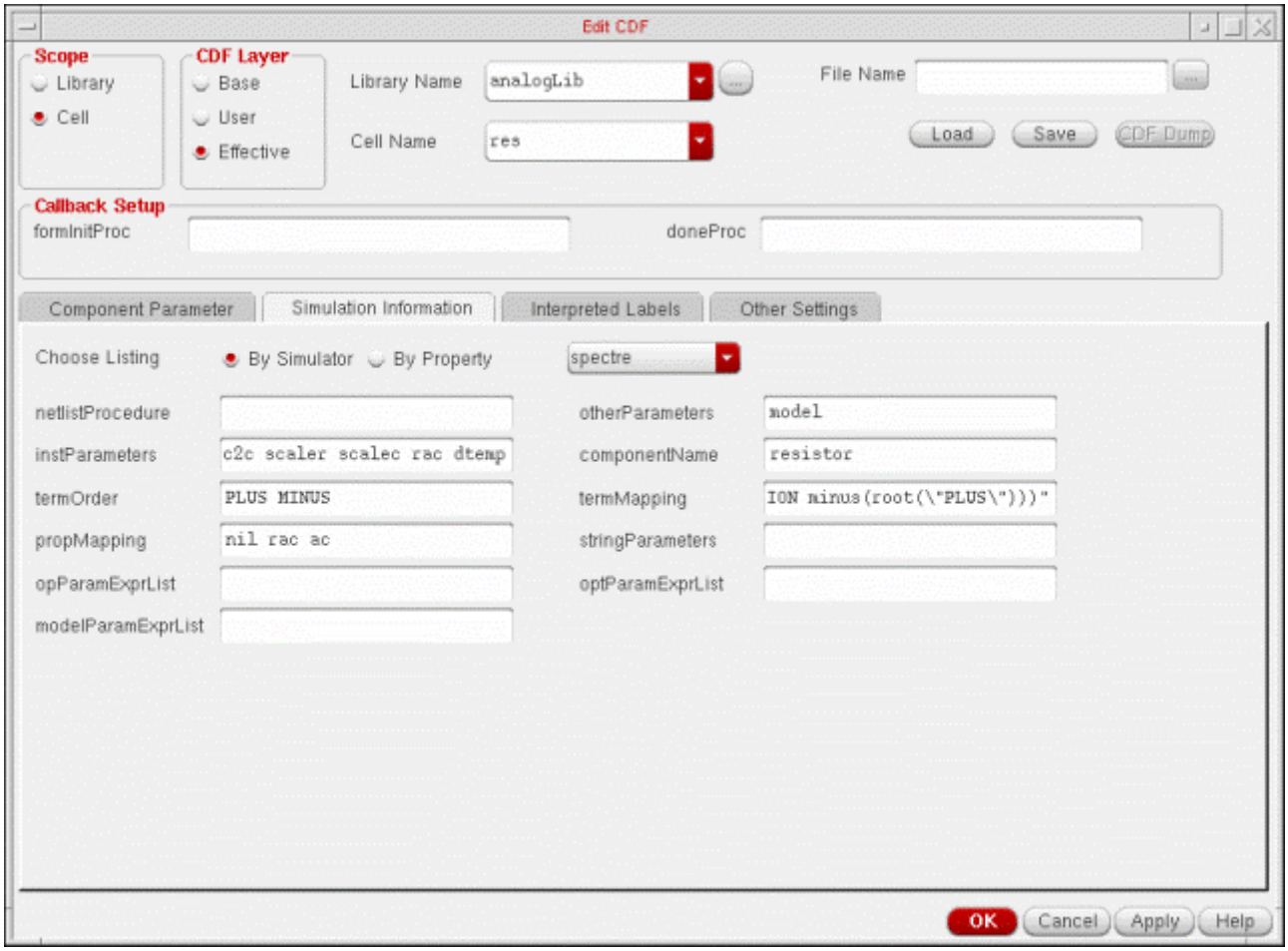
You use the Component Parameter tab of the Edit CDF form to view and edit component parameters. You can also add, delete, and rearrange parameters and edit their attributes.



For information about the Component Parameter tab, see Chapter 3, “Defining Parameters.”

Simulation Information tab

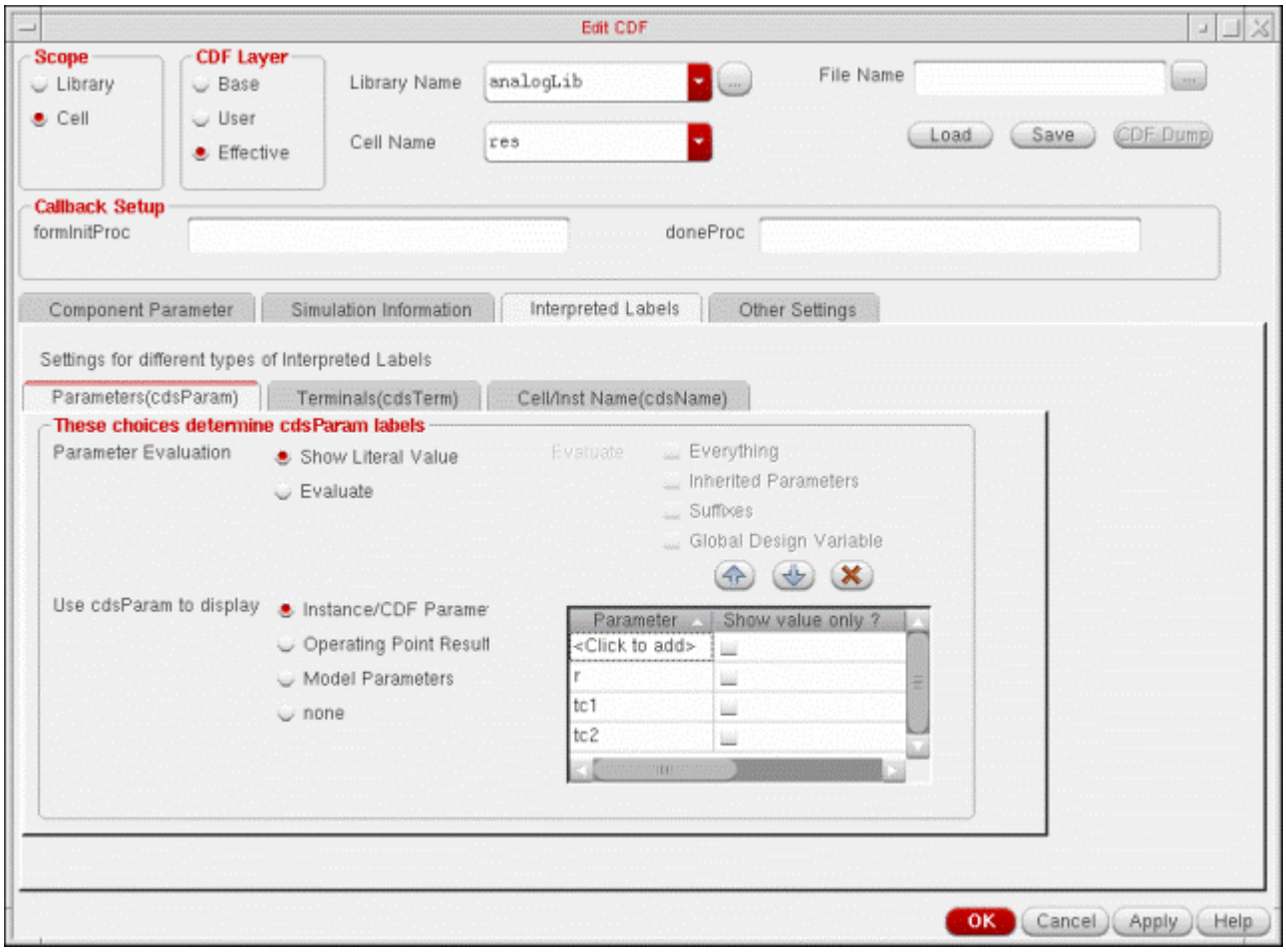
The Simulation Information tab of the form lets you edit simulation information for simulators registered in the software.



For information about modifying the options for simulation information using the Simulation Information tab, see Chapter 4, “Modifying Simulation Information.”

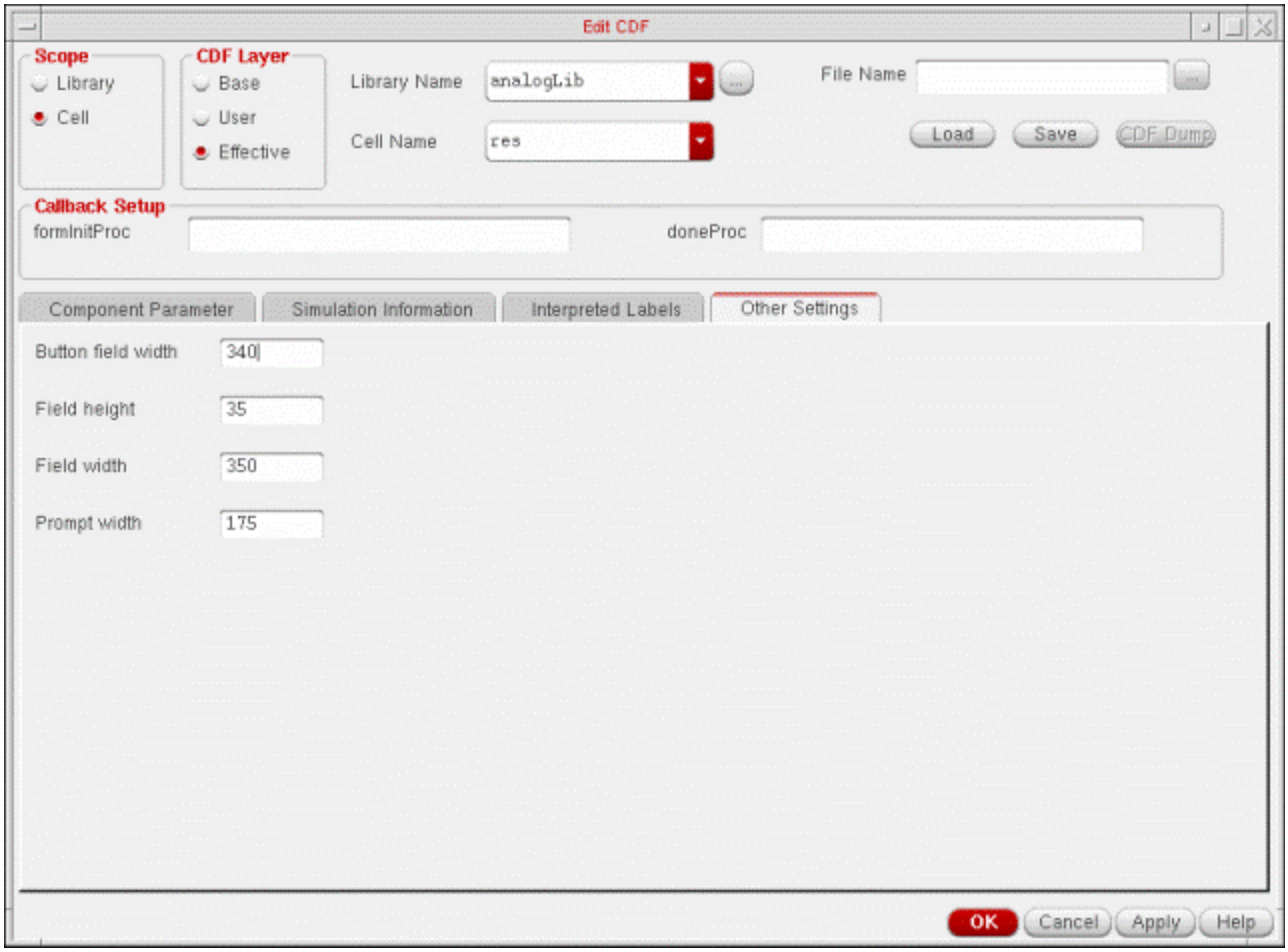
Interpreted Labels Information tab

The Interpreted Labels Information tab of the Edit CDF form lets you edit interpreted label information in the CDF description. Chapter 5, “Specifying Label Information,” describes the options for interpreted labels.



Other Settings

The Other Settings tab of the Edit CDF form lets you edit CDF property fields that are not in any of the previous categories. Chapter 6, “Other CDF Information,” describes the options for other information.

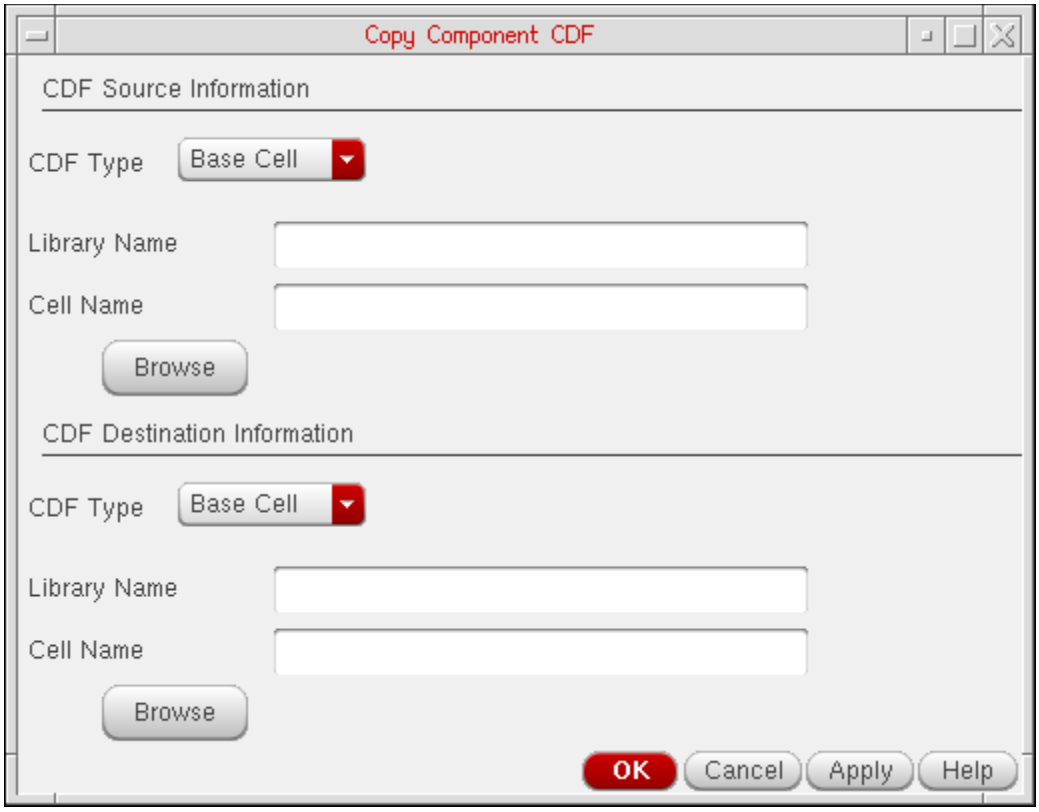


Copy

Tools ➢ **CDF** ➢ **Copy ...**

Copies CDF data from one library or cell to a new library or cell.

Copy Component CDF Form



CDF Source Information

CDF Type is the type of CDF to copy. You can specify either a base-level or a user-level CDF description as the source CDF type. You cannot specify the effective-level CDF description as the source or the destination.

Library Name is the name of the library whose CDF data you want to copy.

Cell Name is the name of the cell whose CDF data you want to copy.

Browse displays the Library Browser.

CDF Destination Information

CDF Type is the type of destination CDF. You can specify either a base-level or a user-level CDF description as the destination CDF type. You cannot use effective-level CDF as a destination CDF type.

Library Name is the name of a library or a library that contains a cell into which you want to copy a CDF description.

Cell Name is the name of the cell into which you want to copy a CDF description.

Browse displays the Library Browser.

Caution

You cannot copy cell CDF data back to the same cell, and you cannot copy a level of CDF data if it has not been created yet. In both cases, the system issues an error message.

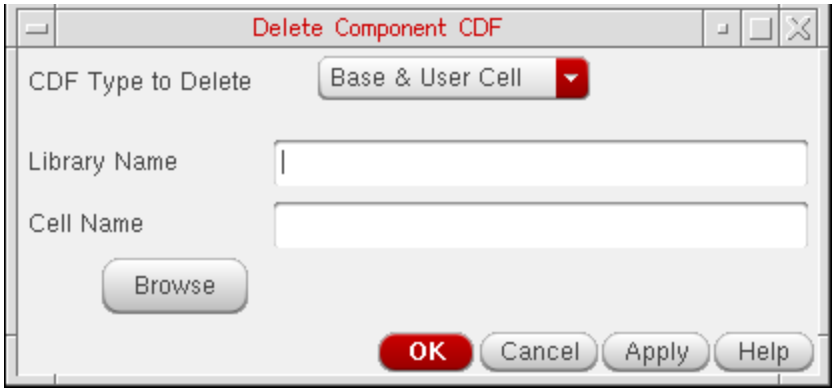
If you copy a cell (component) or library to another cell or library, the CDF data might not be copied. The transfer of CDF data occurs only if the target cell or library never existed and is created in the copy operation. If, however, the target cell or library already exists, or you copy only a cellview (a view of the component), the CDF data of the target cell (or library) remains unchanged. Use the *Copy CDF* command to avoid this problem.

Delete

Tools ➢ **CDF** ➢ **Delete ...**

Removes CDF data from a library or cell.

Delete Component CDF Form



CDF Type to Delete lets you choose the type of CDF to delete.

Library Name is the name of either a library whose CDF description you want to delete or a library that contains a cell whose CDF description you want to delete.

Cell Name is the name of the cell whose CDF description you want to delete.

Browse displays the Library Browser.

Scale Factors

Tools ➢ **CDF** ➢ **Scale Factors ...**

Displays the Units Scaling Factors form, which you can use to set scaling factors for displaying CDF parameters. These abbreviations are used on the form:

f	femto (one quadrillionth)
p	pico (one trillionth)
n	nano (one billionth)
u	micro (one millionth)
m	milli (one thousandth)
c	centi (one hundredth)
k	kilo (one thousand)
M	mega (one million)
G	giga (one billion)

Rounding with Scaling Factors

When a scaling factor is set to a value other than the input value, the output string is rounded based on the following standards:

- If the input value is an integer, no rounding will occur.

```
Input: 1497Output: 1.497K
```

- If the number of decimal digits of the output value is greater than12, the output value will be rounded to maintain no more than12 decimal digits.

```
Input: 1234.12345678901234    Output: 1.234123456789K
Input: 123456.7890123456      Output: 123.456789012346K
Input: 123.4567890123456      Output: 123.456789012346
```

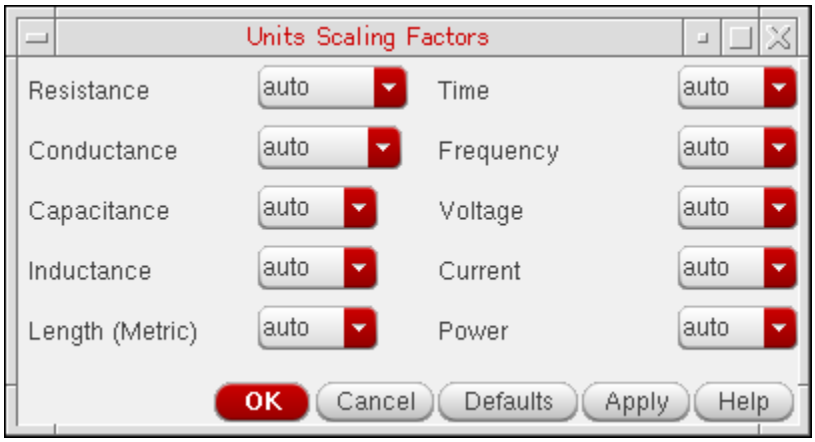
- If the number of decimal digits of the input value is12 or less, the number of decimal digits in the output will be maintained unless it exceeds12.

```
Input: 1497.5                 Output: 1.4975K
Input: 1497.56789             Output: 1.49756789K
Input: 123.12345678987        Output: 1.231234567899K
```

- Trailing decimal zeros will be truncated to the number of decimal zeros in the input.

```
Input: 120000.0              Output: 120.0K
Input: 11000000.00           Output: 11.00M
Input: 11000000.0            Output: 11.0M
Input: 11                    Output: 11000000.0u
```

Units Scaling Factors Form



The following two entries are common to every field on the Units Scaling Factors form:

auto	automatically scales to the most efficient representation
none	uses value as entered, without modifying it

Resistance has a cyclic field for choosing how to show resistance.

t_unitName	t_scaleFactor	Description
`resistance`	*Ohms*	ohms
	kOhms	one thousand ohms
	MOhms	one million ohms

Conductance has a cyclic field for choosing how to show conductance.

t_unitName	t_scaleFactor	Description

`conductance`	*uMhos*	microsiemens
	mMhos	millisiemens
	Mhos	siemens

Capacitance has a cyclic field for choosing how to show capacitance.

t_unitName	t_scaleFactor	Description
`capacitance`	*fF*	femtofarad
	pF	picofarad
	nF	nanofarad
	uF	microfarad
	mF	millifarad
	F	farad

Inductance has a cyclic field for choosing how to show inductance.

t_unitName	t_scaleFactor	Description
`inductance`	*pH*	picohenry
	nH	nanohenry
	uH	microhenry
	mH	millihenry
	H	henry

Length (Metric) has a cyclic field for choosing how to show length.

t_unitName	t_scaleFactor	Description
`lengthMetric`	*uM*	micrometer
	mM	millimeter
	cM	centimeter
	M	meter

Time has a cyclic field for choosing how to show time.

t_unitName	t_scaleFactor	Description
`time`	*pS*	picosecond
	nS	nanosecond
	uS	microsecond
	mS	millisecond
	S	second

Frequency has a cyclic field for choosing how to show frequency.

t_unitName	t_scaleFactor	Description
`frequency`	*Hz*	hertz
	kHz	kilohertz
	MHz	megahertz
	GHz	gigahertz

Voltage has a cyclic field for choosing how to show voltage.

t_unitName	t_scaleFactor	Description
`voltage`	*uV*	microvolts
	mV	millivolts
	V	volts
	kV	kilovolts

Current has a cyclic field for choosing how to show current.

t_unitName	t_scaleFactor	Description

`current`	*pA*	picoamperes
	nA	nanoamperes
	uA	microamperes
	mA	milliamperes
	A	amperes

Power has a cyclic field for choosing how to show power.

t_unitName	t_scaleFactor	Description
`power`	*uW*	microwatts
	mW	milliwatts
	W	watts

SKILL Function

You can use the following Cadence SKILL language commands to set scale factors.

```
cdfGetUnitScaleFactor( t_unitName) => t_scaleFactor
cdfSetUnitScaleFactor( t_unitName t_scaleFactor) => t / nil
```

For example, to set lengthMetric to m (millimeters), use the following command:

```
cdfSetUnitScaleFactor("lengthMetric" "m")
```

You can use abbreviations listed earlier instead of specifying the complete scale factor. For example, to change the scale factor for resistance to *MOhms*, enter:

```
cdfSetUnitScaleFactor("resistance" "M")
```

To display the current scale factor for *power*, enter:

```
cdfGetUnitScaleFactor("power")
```

The system returns the present value, such as *auto.*

You can also set the *t_scaleFactor* to *auto* or *none* using the `cdfGetUnitScaleFactor` SKILL command. In SKILL expressions only, you can use the following ISO1000 standard units in scale factors.

Symbol	Name	Factor
Y	yotta	10^{24}
Z	zetta	10^{21}
E	exa	10^{18}
P	peta	10^{15}
T	tera	10^{12}
h	hecto	10^{2}
da	deca	10
d	deci	10^{-1}
centi	c	10^{-2}
a	atto	10^{-18}
z	zepto	10^{-21}
y	yocto	10^{-24}

To display the Units Scaling Factors form and edit the scale factors, use the following SKILL function.

cdfEditScaleFactors

```
cdfEditScaleFactors()
     => t / nil
```

Displays the Units Scaling Factors form which can be used to set scaling factors for displaying CDF parameters.

SKILL CDF Functions

You can use SKILL functions to create, view, or modify CDF information. Chapter 8, “CDF SKILL Summary,” describes the SKILL functions you can use to do the same data entry operations you perform with the Edit CDF form.

To view the SKILL for a CDF description, use the `cdfDump` function.

```
cdfDump(
"libName"
filename
?cellName "cellName"
?level {’base | ’user}
?edit b_edit
)
```

A specific example is:

```
cdfDump("analogLib" "edtmp" ?cellName "nfet" ?level ’base ?edit t)
```

If you specify the *`libName`* only, *`cdfDump`* writes only the library CDF data to the file specified in *`filename`*. The other arguments are optional. You must always specify a *`libName`* and a *`filename`*.

The `?cellName` argument dumps the CDF description from that cell.

With the `?level` argument, you can choose base- or user-level CDF. The default is `base`.

The `?edit` option lets you automatically load the dump file into a temporary editor window. The default is `nil`.

You can specify a text editor by adding a line to your `.cdsinit` file. The following example uses the `vi` editor.

```
editor = "xterm -g 80x35+30+30 -e vi"
```

If you have write permission for a cell, you can reload the CDF description with the following command in the Command Interpreter Window (CIW):

```
load("filename")
```

If you have the *skillDev* software, you can get a list of all the available CDF functions by typing the following command in the CIW:

```
listFunctions("cdf")
```

If you want to change the CDF description of a component but do not have the appropriate permissions, you can copy the cell to one of your own libraries. Copying a cell to a new cell also copies its CDF description, which you can then edit. Alternatively, you can change the user-level CDF description.

To create a CDF description for a new cell, you can first dump the CDF description from a similar cell, change the cell or library name, then load the file as described in this section. If the library and cell you specified exist, a new CDF description is created.

Saving User-Level CDF Information

User-level CDF information is not written to your system disk. When you exit the Cadence software, all user-level CDF information that you have not specifically saved is lost. If you want to save your user-level data, you must write it to a file.

If you are using the Edit CDF form in the Cadence Analog Design Environment, you can save any CDF description that you are working on with the *File Name* field and its associated buttons.

In the *File Name* field, enter the name of the file where you want to save the CDF data, and click *Save*.

1. To retrieve this CDF information, open the Edit CDF form again.
2. Select the name of the library, cell, and CDF Layer (User).
3. Type the filename in the *File Name* field, and click *Load*.

Using SKILL

If you are using SKILL, use `cdfDump` to save the user-level CDF data and `load` to restore it.

1. Type the following SKILL command:

   ```
   cdfDump("lib_name" "filename" ?cellName "cell_name"
       ?level 'user ?edit t)
   ```

   An editor window appears, showing the file specified in *`filename`*.

2. Save the contents of this new file.
3. To restore the user-level CDF data, type

   ```
   load("filename")
   ```

The `load` command reads the information in the file to determine which library, cell, and level of CDF description to access. It then executes the SKILL functions contained in the file, replacing the saved user-level data in the correct library and cell.

Complex Pole Example

To look at the CDF information for this component, you need the functional library, which is in the following location:

`<your_install_dir>/etc/cdslib/artist/functional`

Make sure that your `cds.lib` file includes the definition of this library.

Using the Edit CDF Form

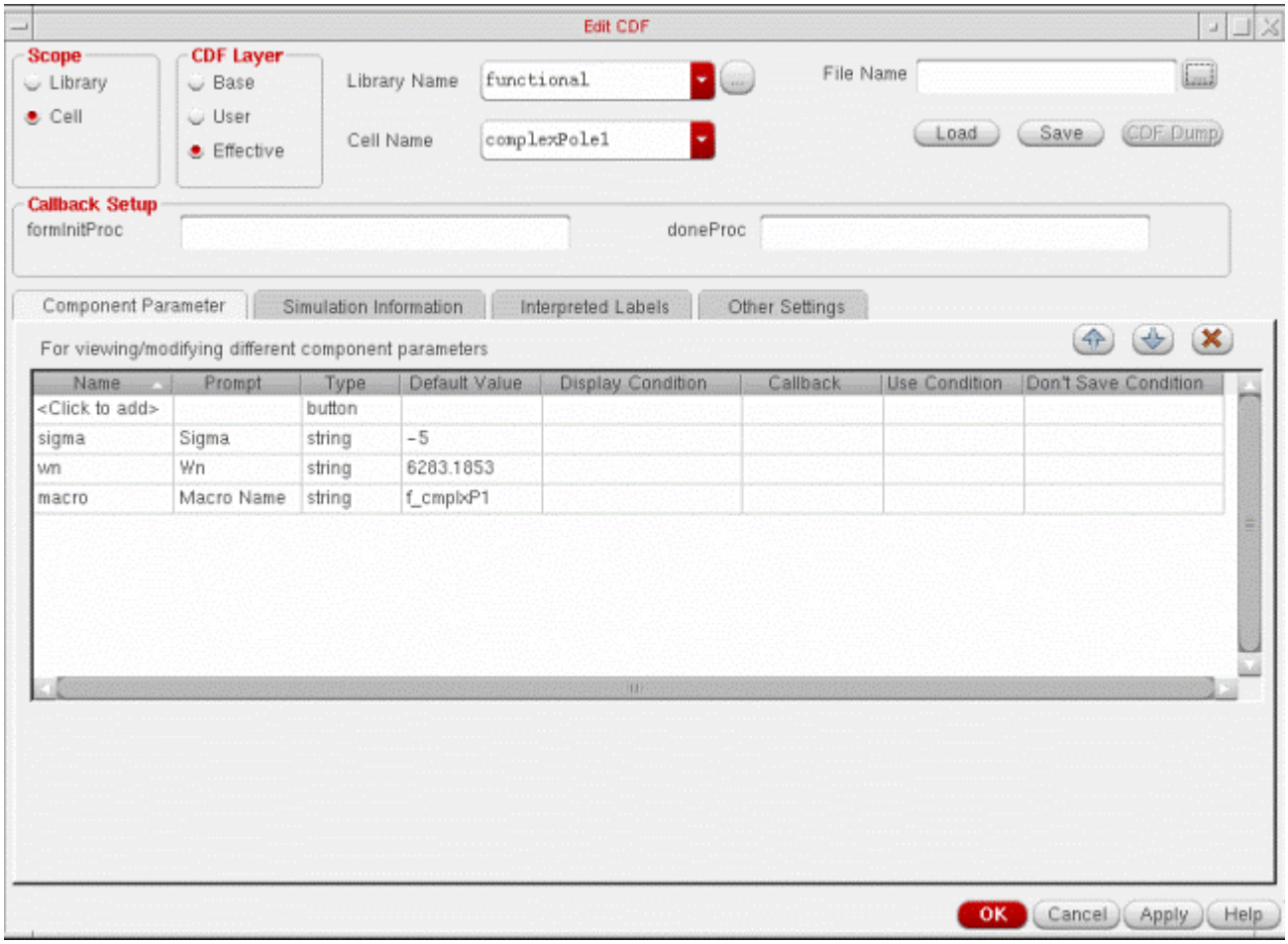
Follow these steps to edit the CDF of the `complexPole1` cell in the functional block library:

1. From the CIW menu banner, choose *Tools – CDF – Edit*.

   The Edit CDF form appears.

2. In the *Scope* group box, select *Cell*.
3. In the *Library Name* cyclic field, select the library *functional*.
4. In the *Cell Name* cyclic field, select the cell *complexPole1*.

   The CDF description for the *complexPole1* component appears in the form.



You should see a header and four tabs, each with information specific to the complex pole cell.

Using the header, you can save the current CDF descriptions to a named file or load another CDF description onto the named cell from a file. This is useful if you want to see the results of changing certain CDF properties. You can store the alternatives in different files and load them as required. You can also save the CDF data from one cell and then load (transfer) it onto another cell.

At the end of Chapter 3, “Defining Parameters,” you examine the Component Parameter tab in detail.

NFET Example

In the complex pole example, you examine an existing CDF description. In the NFET example, you create a new cell and a new CDF description.

Using the Edit CDF Form

When you copy a cell to another cell, all views and the CDF descriptions are copied if the destination cell did not previously exist. Copy the *nbsim* component from the analog library to create the basis for the new *nfet* component. You can then edit settings for parameters, simulation, labels, and other information in the CDF description of the *nfet* component.

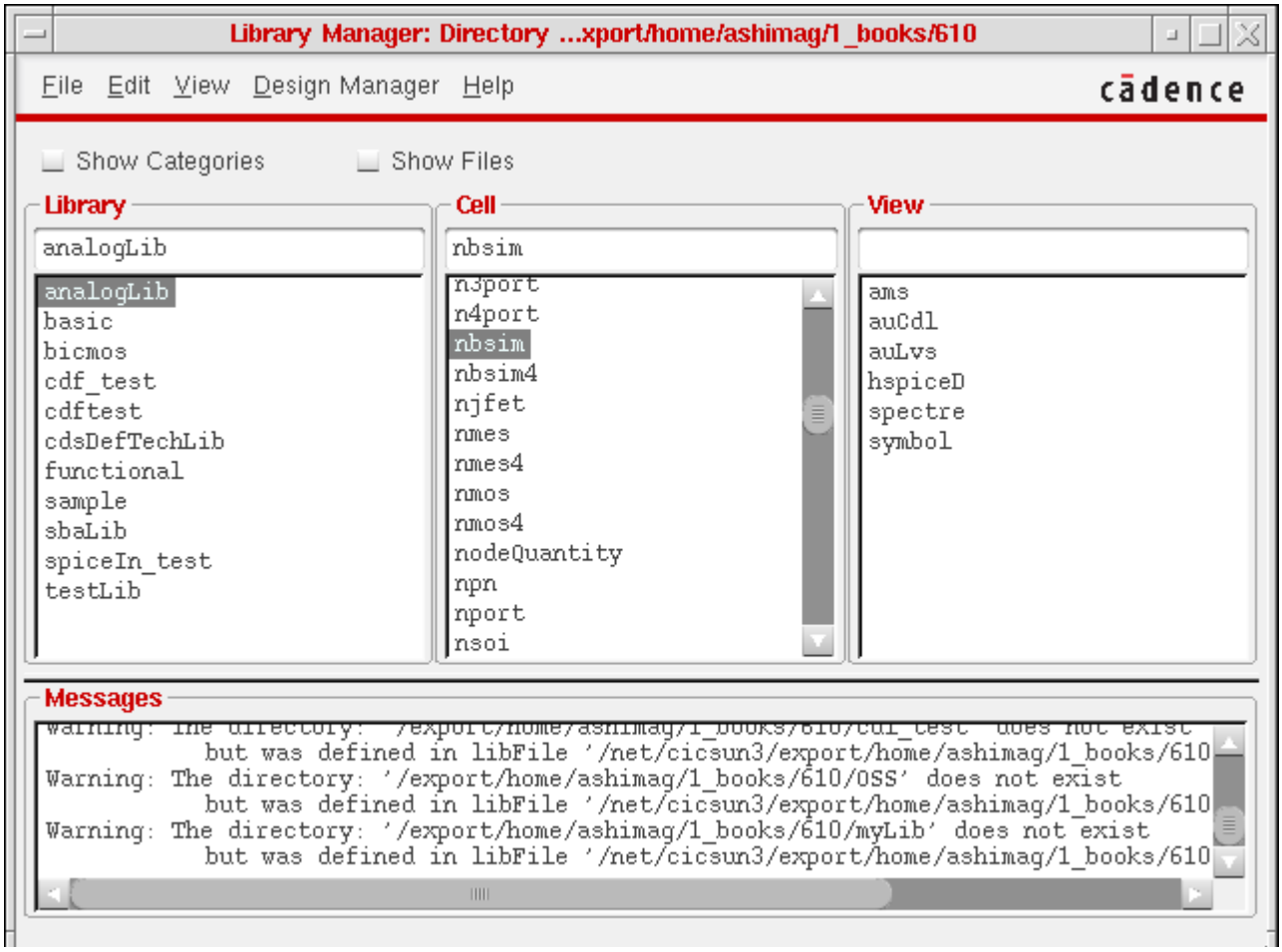
1. Create a library called *bicmos*.

   Make sure that you include this library in your `cds.lib` file.

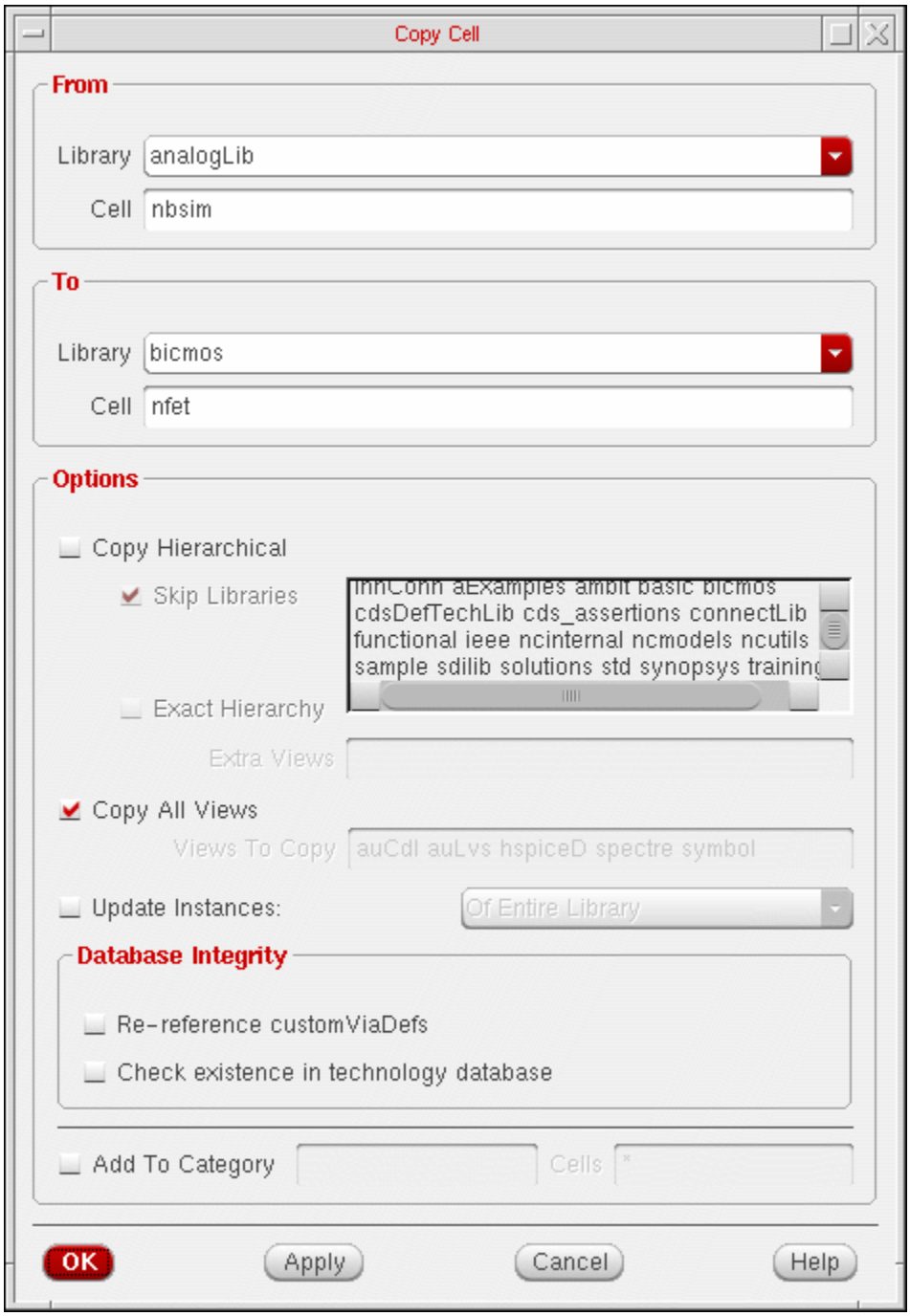
2. In the CIW, open the Library Manager by choosing *Tools – Library Manager*.

3. Select *analogLib* to display its components (cells).

4. Scroll down the cell list and select *nbsim*.

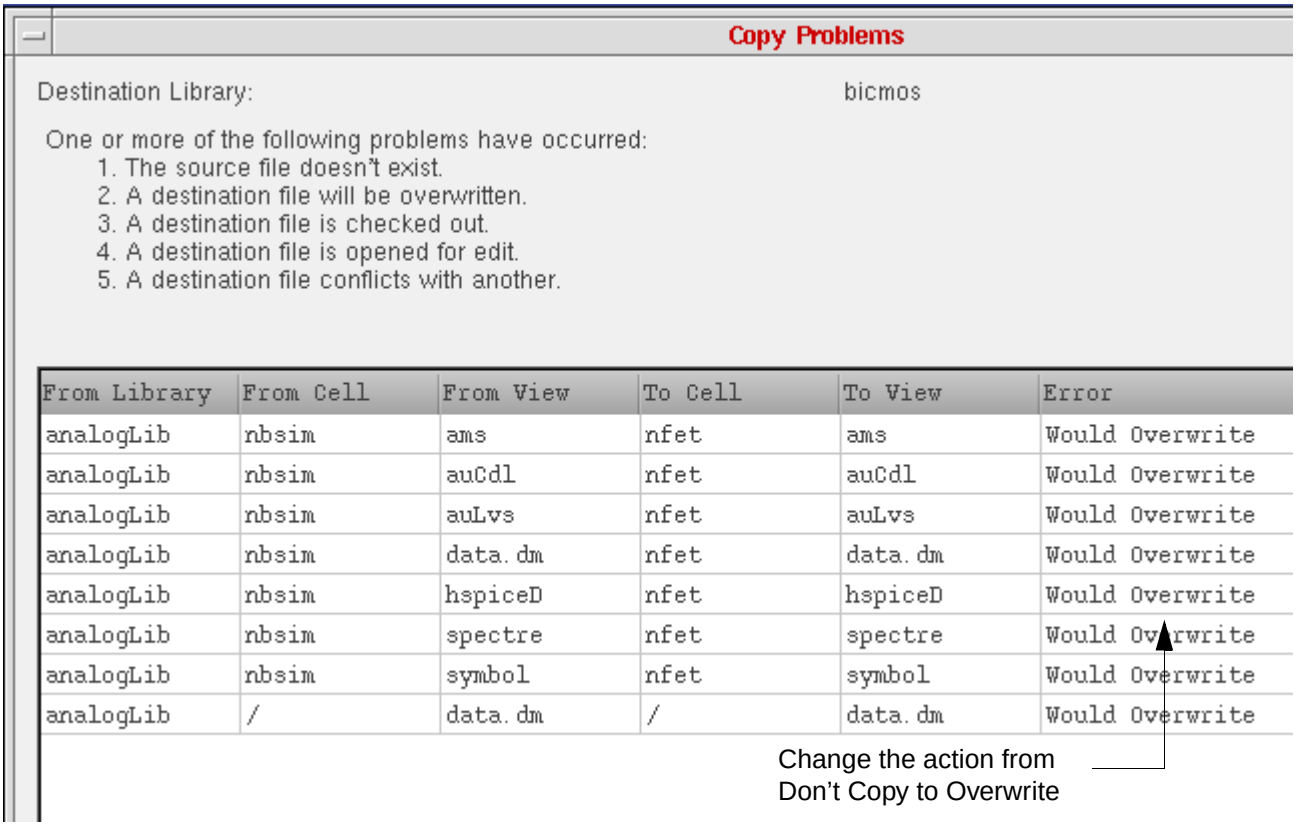


5. Choose *Edit – Copy* and select *bicmos* from the *To* library cyclic field and specify the target cell as *nfet*.



6. Click *OK* to make a copy of *nbsim* in the *bicmos* library under the cell name of *nfet*.

7. In the *Copy Problems* dialog box change the *Don't Copy* action to *Overwrite* from the Action cyclic field for each view type.



From Library	From Cell	From View	To Cell	To View	Error
analogLib	nbsim	ams	nfet	ams	Would Overwrite
analogLib	nbsim	auCdl	nfet	auCdl	Would Overwrite
analogLib	nbsim	auLvs	nfet	auLvs	Would Overwrite
analogLib	nbsim	data.dm	nfet	data.dm	Would Overwrite
analogLib	nbsim	hspiceD	nfet	hspiceD	Would Overwrite
analogLib	nbsim	spectre	nfet	spectre	Would Overwrite
analogLib	nbsim	symbol	nfet	symbol	Would Overwrite
analogLib	/	data.dm	/	data.dm	Would Overwrite

8. Alternatively from the CIW, choose *Tools – CDF – Copy*.

   The Copy Component CDF form opens.

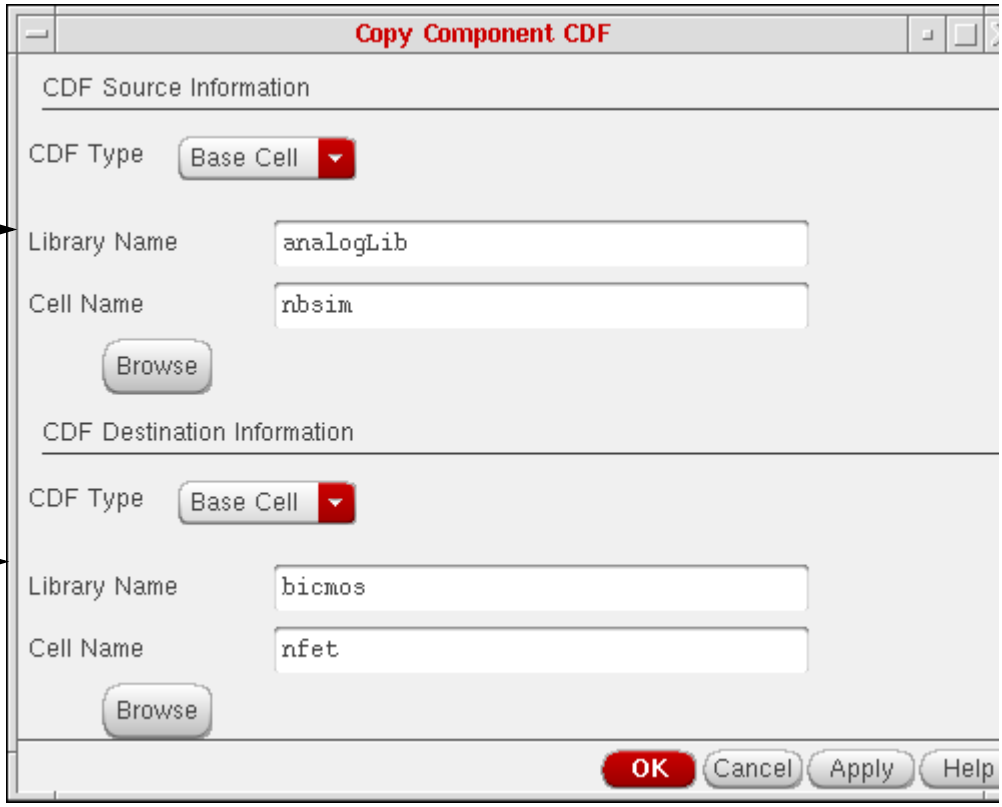
9. Type the *analogLib* library and the *nbsim* cell in the CDF Source Information section.

10. Type the name of your destination library (*bicmos*) and the cell name (*nfet*) in the CDF Destination Information section.

Press the *Tab* key at the end of each entry.

The source cell is entered automatically when browsing

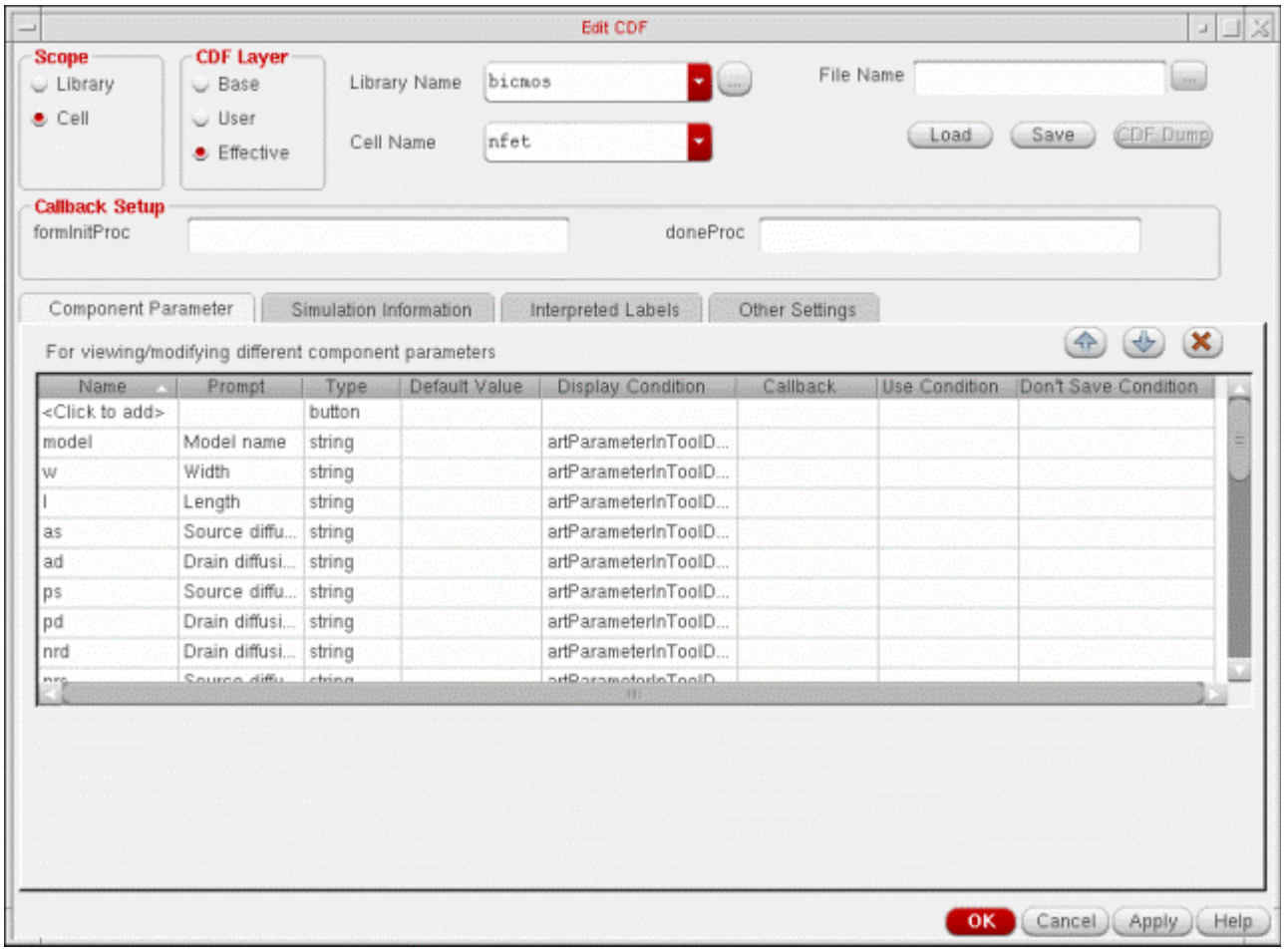
Enter the name and library of the destination cell.



11. Click *Apply*.

You can look at the CDF description of *nfet* using the Edit CDF form, as you did in the complex pole example in this chapter.

➤ From the CIW, choose *Tools – CDF – Edit*.



In Chapter 3, “Defining Parameters,” you will change some of the parameter attributes to make *nfet* a different part from *nbsim*.

Using SKILL

The following SKILL function copies the base-level CDF description of *nbsim* in *analogLib* to *nfet* in *bicmos*.

```
cdfCopyCDF( dest_cell "baseCellData" srcCDF )
```

This is possible only after you have performed these assignments, in order:

```
    source_cell = ddGetObj( "analogLib" "nbsim" )
    srcCDF = cdfGetBaseCellCDF( source_cell )
    dest_cell = ddGetObj( "bicmos" "nfet" )
    if( destCDF = cdfGetBaseCellCDF(dest_cell) then
        cdsDeleteCDF( destCDF )
    )
```

If the `destCDF` already has a CDF description of the specified type, a warning message will be displayed in the CIW indicating that the Base/User CDF already exists. In this case, the CDF copy function will not be performed.

Refer to Appendix C, “NBSIM Transistor CDF SKILL Description,” for an example of the SKILL CDF description of the *nbsim* cell.

3

Defining Parameters

Overview

This chapter describes how to create cell parameters and set parameter attributes.

Each cell has a set of one or more parameters that describe the function of the cell. At any one time, a parameter has one value and a set of attributes. Unlike the value, attributes impose features and limitations on the parameters themselves, independent of their values. For example, the Boolean attribute means that a parameter can have only one of two values. A float attribute requires that the value of the parameter be a floating-point number.

Another attribute can determine whether the parameter value is displayed on a form or in the schematic.

```
Cell
    ├ parameter
    ├ parameter
    ├ parameter
    └ parameter — value
          ├ attribute
          ├ attribute
          └ attribute
```

The parameters section of a Component Description Format (CDF) describes what parameters the component has, and the attributes of a component's parameters. You can describe the value of a parameter using long expressions based on the Analog Expression Language (AEL). These expressions can include mathematical operations and functions that call the value of other parameters. Attribute values are limited to very specific values.

When a cell is instantiated into a design, the parameters of that cell become properties.

The difference between a parameter and a property is the following:

- A parameter is a characteristic of a component that has special meaning to the component. Different instances of the same component do not always use the same parameters and almost never use the same values for their parameters.
- A property is how a component parameter is viewed by the Cadence® database when it deals with each instance.

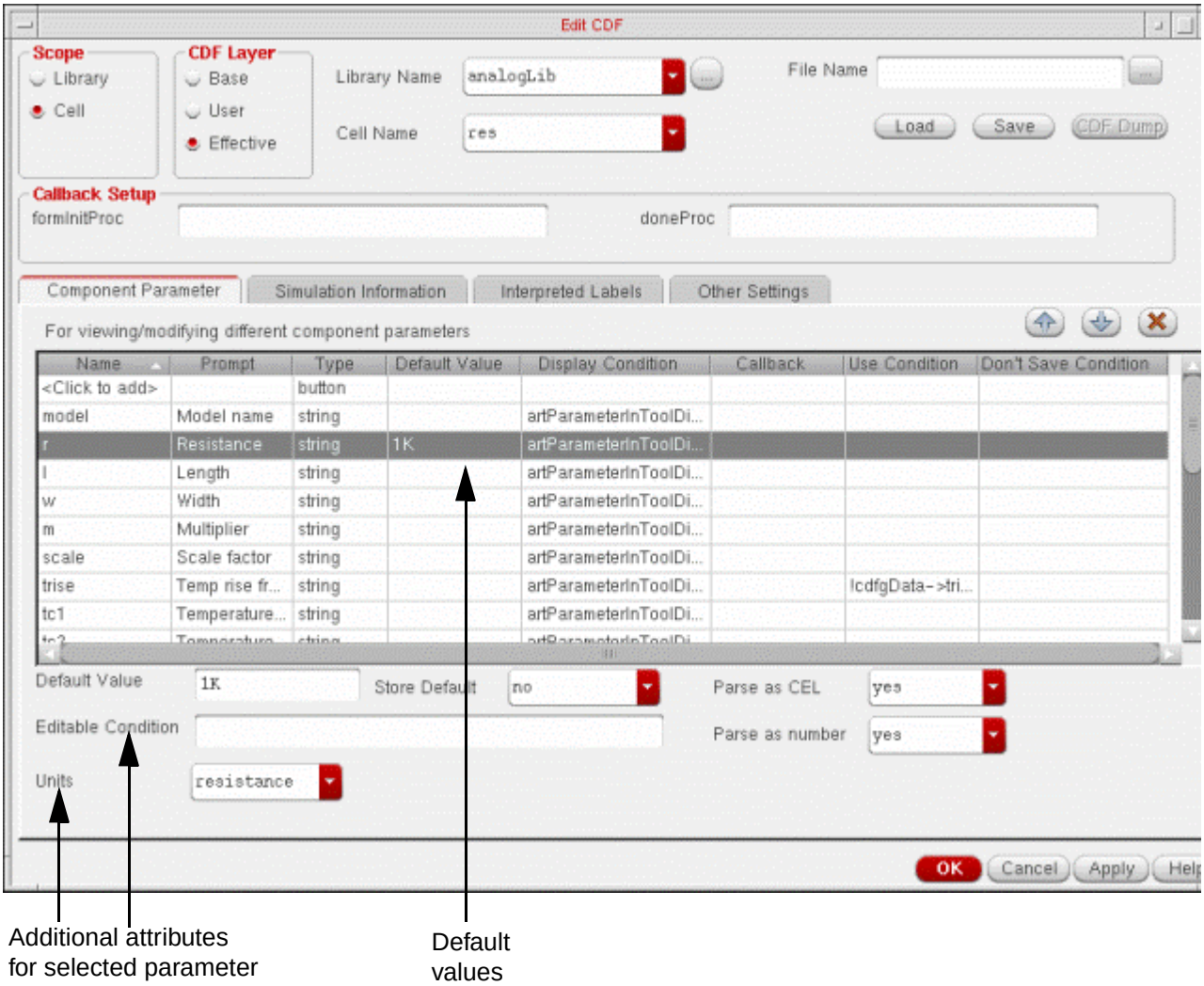
You can set up a hierarchy of CDF descriptions, one overriding the other, because you might want to override specific attributes of some parameters. For example, if you define the *xyz*

parameter on a base-level CDF, you can define the same *xyz* parameter on the corresponding user-level CDF. Because the user-level description overlays the base-level description, the user-level attributes override the base-level attributes. However, certain parameter attributes, such as the parameter name, cannot be overridden.

Note: Because user-level CDF is specific to a user and not a library or cell, simulation results can be different for different users. The use of user-level CDF is therefore discouraged.

Component Parameters

The Component Parameter tab of the Edit CDF form lists all of the parameters defined for the selected CDF description. (Refer to Chapter 2, "CDF Commands," to learn how to select libraries and cells.) Use the scroll bar on the right side of the tab to view more parameters.



The parameter information is displayed in the columns described in the Table 3-1. When you select a parameter, additional attributes can be specified for the parameter in the fields that appear in the bottom part of the Component Parameter tab. These fields are described in Table 3-2 on page 58.

The buttons on the top right of the Component Parameter tab allow you to move the selected parameter relative to other parameters, or delete the selected parameter. These buttons are described in Table 3-3 on page 63.

Table 3-1 Columns in the Component Parameter Tab

Column	Description
Name	Displays the parameter name. All parameters must have a name defined, and parameter names cannot begin with a number. Sometimes this name is dictated by the application that uses the parameter. To add a new parameter double-click where it says <*click to add*>, type the parameter name, then press *Enter*.
Prompt	Displays the prompt for the parameter. It is useful if you become familiar with parameters from their details in other forms that use the prompt rather than the name.
Type	Displays the data type of the parameter. You need to specify data type for every parameter. Valid data types are *string*, *int* (integer), *float* (floating-point number), *radio* (radio button), *cyclic* (cyclic field), *boolean, button*, and *netSet*.
Default Value	Displays the default value specified for the parameter. **Note:** You cannot enter or modify the default values displayed in this column. To specify the default value for a parameter, select the parameter and enter the default value in the *Default Value* field that appears in the bottom part of the Component Parameter tab.
Display Condition	Determines if this parameter is displayed in forms that display CDF parameters, such as the Edit Object Properties form or the Add Instance form. You must enter *t*, *nil*, or a SKILL expression that evaluates to *t* or *nil* in this field to determine if this parameter is to be displayed. If the field evaluates to non-*nil* (the default), the parameter is displayed. If the field evaluates to *nil*, the parameter is not displayed.

Table 3-1 Columns in the Component Parameter Tab

Column	Description
Callback	Specifies a SKILL routine to be executed *whenever the value of the parameter changes*. The value of this optional field must be a string. Your entry can be the entire function to be executed or a call to a function that is defined elsewhere. If you do not enter anything in this field, the system assumes that there is no callback. The CDF parameter callback is primarily a GUI based callback. GUI based callbacks occur when you modify the values in the parameter form fields. A GUI based callback is active when the CDF parameters are displayed in the Add Instance form or the Edit Object Properties form when you use the Create Instance or Edit Properties commands. For more information on callbacks, refer to Chapter 7, "Writing Callbacks."
Use Condition	Determines if the parameter is to be used. Often, you can enter Cadence SKILL language expression that evaluates to *t* or *nil* in this field to determine if this parameter is applicable. When the field evaluates to *nil*, the system never displays the parameter. When it evaluates to non-*nil* (the default), then based on the value of **Display Condition**, the system will display the parameter.
Don't Save Condition	Determines if the parameter value is to be saved on the instance. This attribute is for programming use only. Typically, you should set this field to *nil* and not use it. If the field evaluates to *nil* (the default), the parameter value is saved as a property on the instance. If the field evaluates to non-*nil*, the parameter value is not saved as a property on the instance. *Caution* ***The Don't Save Condition attribute overrides the Store Default setting. If Store Default is yes and Don't Save Condition is yes, not even the default property is saved.***

The following fields are displayed in the bottom part of the Component Parameter tab when you select a parameter in the Component Parameter tab. The display of these fields depend

on the data type of the parameter. For more information, see Attributes and Parameter Types on page 64.

Table 3-2 Fields in the bottom part of the Component Parameter Tab

Field	Description
Default Value	Specifies the default value for the selected parameter. Most parameters must have a default defined. The default value depends on the data type for this parameter. Parameter values can be long mathematical expressions with variables and functions of other parameters. **Note:** A default value is mandatory for parameters of type `int`, `float`, `radio`, `cyclic` and `netset`. If you do not specify a default value for parameters of these types, an error message in the CIW displays the list of parameters for which a default value is not specified, and these parameters are not saved in the CDF if you click *OK* or *Apply* in the Edit CDF form. If you do not want the Edit CDF form to be closed without saving these parameters in the CDF, set the `cdfAllowOKApplyPopups=t` environment variable in your `.cdsenv` file or `.cdsinit` file, or enter `cdfAllowOKApplyPopups=t` in the CIW. If this environment variable is set, a popup message displays the list of parameters for which a default value is not specified, and you cannot close the Edit CDF form until you specify a default value for these parameters.

Table 3-2 Fields in the bottom part of the Component Parameter Tab

Field	Description
Store Default	Specifies whether to store the default value of a parameter as a parameter attribute on the instance. All tools based on the Cadence Virtuoso Analog Design software use the CDF description to find default values if there is no property on an instance. If set to *no* or *don't use*, then the default value that you set for a parameter is not preserved. In such a case, if you modify the default value of a parameter, then the change is reflected in all existing instances as well as new instances of a component. When the default value of the CDF parameter changes, all the instances including the ones already instantiated, are updated with the new default value of the CDF parameter. To see the change in an open window you must choose *Window – Redraw* from the Cadence menu. If set to *yes,* then a parameter attribute that stores the default value of the parameter is added on the instance. Later if the default value for the parameter is modified, then those instances that are already instantiated, retain the old default value of the parameter. Only new instantiations have the new default value of the parameter. One disadvantage of setting this attribute to *yes* is that if the default value of a parameter changes, the existing instances that use the default value do not automatically change to the new default value. That is, they retain the old default value. By default, *Store Default* is set to *no*.

Table 3-2 Fields in the bottom part of the Component Parameter Tab

Field	Description
Parse as CEL	Specifies whether the parameter is processed as a CDF Expression Language (CEL) expression. Use this attribute only for string type parameters. If *Parse as CEL* is *yes*, the parameter is processed as a CEL expression. (Expressions such as *iPar(x)* or *pPar(x)*, indicating inheritance from an instance parameter or a parent parameter, are processed as CEL expressions.) If *Parse as CEL* is *no* (the default), the parameter is not processed as a CEL expression. It is taken literally.
Parse as number	Specifies whether this parameter can evaluate to a floating-point number. Use this attribute only for *string* type parameters that contain numeric data. If *Parse as Number* is *yes*, the system converts the string to a floating-point number, converts the floating-point number to the most efficient notation for execution, and reconverts the floating-point number into a string. If *Parse as Number* is *no* (the default) or *don't use*, the system does no conversion. Use this setting if the parameter to be defined is a literal that contains numeric characters, such as the name of a file or a model. *Parse as Number* must be used when *Parse as CEL* is *yes*.
Editable Condition	Determines if this parameter can be edited in forms that display CDF parameters, such as the Edit Object Properties form or the Add Instance form. Often, you can enter a SKILL expression that evaluates to *t* or *nil* in this field to determine if the parameter is editable. If the field evaluates to non-*nil* (the default), the parameter is editable. If the field evaluates to *nil*, the parameter is not editable. (If not editable, the parameter is dimmed so that you can see the value but you cannot edit it.) This field is valid only for *string*, *int*, and *float* data types.

Table 3-2 Fields in the bottom part of the Component Parameter Tab

Field	Description
Units	Determines the unit suffix and scale factor to use when displaying the parameter value. Use this attribute only with *string* type parameters where the response to *Parse as Number* is *yes*. (This specification has no effect if you set it for other parameter types.) The units attribute must be one of the following: resistance, capacitance, inductance, conductance, time, frequency, power, powerDB, lengthMetric, lengthEnglish, angle, voltage, current, or temperature.

Table 3-2 Fields in the bottom part of the Component Parameter Tab

Field	Description
Choices	The space- or comma- separated list of selections for a *cyclic* or *radio* data type. This attribute does not apply to other types of parameters. How to specify choices depends on two cases: ■ If each choice is a single word, you can separate the choices with spaces. ■ If any choice is a group of words, you must separate each choice with a comma (,). (When using commas, do not leave extra spaces between choices because these spaces become part of the choice value.) A typical entry in the form field might be `choice 1,choice 2,choice 3` Notice that there is no blank between the number 1 and the comma, or between the comma and the *c* in *choice*. **Note:** The *Choices* field is mandatory for parameters of type `radio` and `cyclic`. If you do not specify the choices for parameters of these types, an error message in the CIW displays the list of parameters for which the choices are not specified, and these parameters are not saved in the CDF if you click *OK* or *Apply* in the Edit CDF form. If you do not want the Edit CDF form to be closed without saving these parameters in the CDF, set the `cdfAllowOKApplyPopups=t` environment variable in your `.cdsenv` file or `.cdsinit` file, or enter `cdfAllowOKApplyPopups=t` in the CIW. If this environment variable is set, a popup message displays the list of parameters for which the choices are not specified, and you cannot close the Edit CDF form until you specify the choices for these parameters.

Table 3-3 Buttons in the Component Parameter Tab

Button	Description
	**Move Up** lets you change the sequence of existing parameters. You can move a parameter relative to other parameters in the Component Parameters section of the Edit CDF form. Commands that use CDF data display CDF parameters in the same order as the Component Parameter tab. To avoid unnecessary scrolling, place frequently used parameters at the top.
	**Move Down** lets you change the sequence of existing parameters. You can move a parameter relative to other parameters in the Component Parameters section of the Edit CDF form:
	**Delete** lets you delete the selected parameter from the CDF. The only way you can undo a deletion is to cancel all of your edits.

Note: The CDF description does not change until you click *Apply* or *OK* on the Edit CDF form.

Attributes and Parameter Types

Certain attributes are meaningful for some parameter types but not for others. For example, the *editable* attribute is meaningful for *string*, *integer*, and *float* types, but not for *radio*.

The following table shows the attributes that are required (req), optional (opt), and unused for each data type. The *use* attribute is for programming use only (prog). You should set the *dontSave* attribute to the default value *nil* for all parameter types.

Attribute in UI[1]	Attribute in SKILL[2]	String	Integer	Float	Radio	Cyclic	Boolean	Button
Type	paramType	req	req	req	req	req	req	req
Parse as number	parseAsNumber	opt	unused	unused	unused	unused	unused	unused
Units	units	opt	unused	unused	unused	unused	unused	unused
Parse as CEL	parseAsCEL	opt	unused	unused	opt	opt	unused	unused
Store Default	storeDefault	opt	opt	opt	opt	opt	opt	unused
Name	name	req	req	req	req	req	req	req
Prompt	prompt	req	req	req	req	req	req	req
Choices	choices	unused	unused	unused	req	req	unused	unused
Default Value	defValue	req	req	req	req	req	req	unused
Use Condition	use	prog	prog	prog	prog	prog	prog	prog
Display Condition	display	opt	opt	opt	opt	opt	opt	opt
Don’t Save Condition	dontSave	nil	nil	nil	nil	nil	nil	nil
Editable Condition	editable	opt	opt	opt	unused	unused	unused	unused
Callback	callback	opt	opt	opt	opt	opt	opt	opt

1. Attribute name displayed in the Edit CDF form.

2. Attribute named used in SKILL.

Editing Parameters

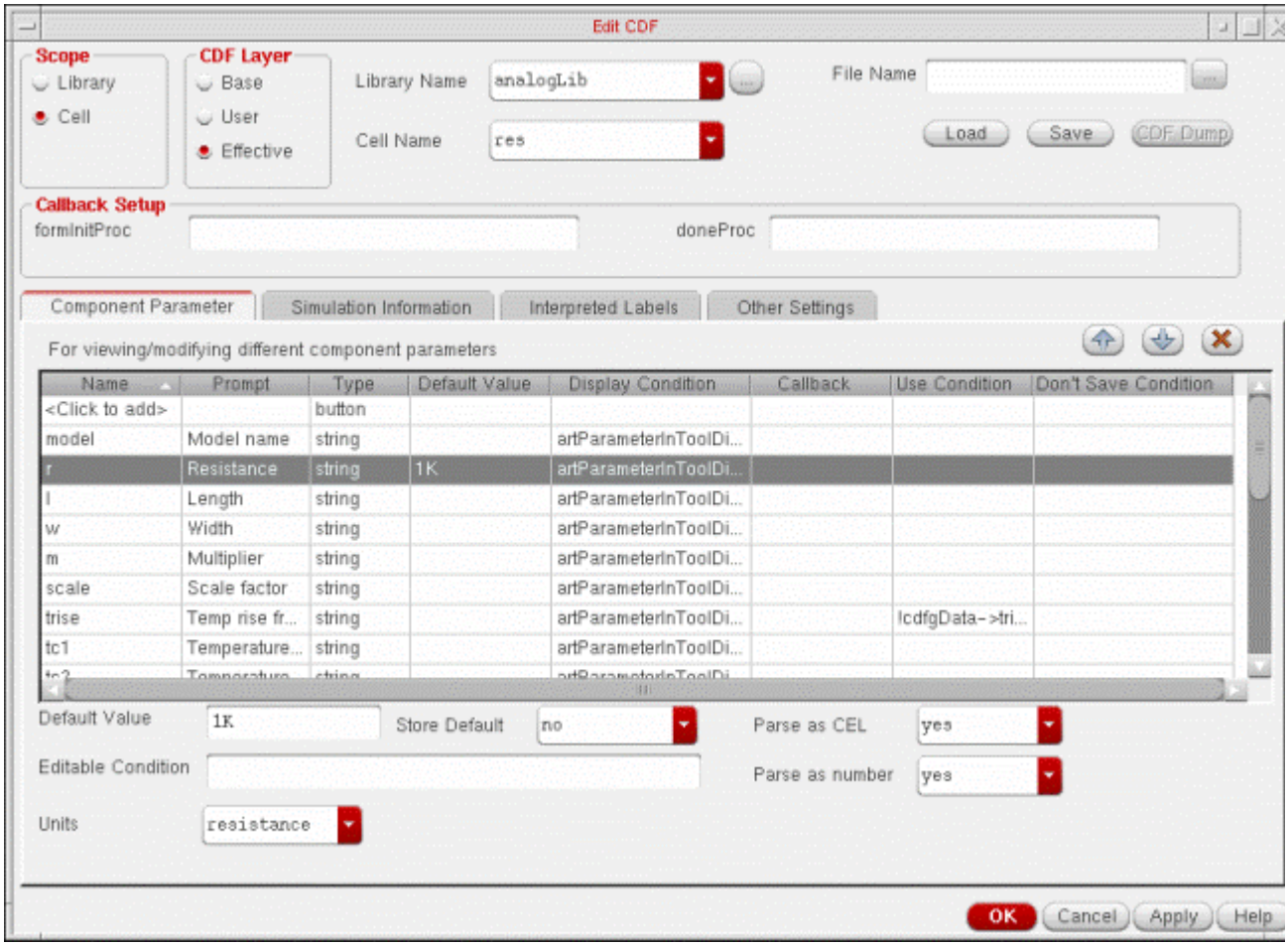
All the parameter attributes descriptions in the previous section apply whether you are using SKILL or the Edit CDF form. The following sections describe how you edit parameters using either method.

Using the Edit CDF Form

You use the Component Parameter tab in the Edit CDF form to edit parameter attributes to test variations in your design. You can also create a new parameter by editing an existing parameter and changing its name. This way, you can use some or all of the parameter attributes from the original parameter.

1. Click the Component Parameter tab in the Edit CDF form.
2. Select the parameter you want to edit.

The attributes for the parameter are displayed in the fields that appear in the bottom part of the Component Parameter tab.



3. Change the parameter attributes.
4. Click *Apply* to enter the changes.

Using SKILL

You can use SKILL functions to create and modify parameters and their attributes. The following SKILL functions create the parameters for the sample CDF description represented here by the variable *cdfDataId*.

The first function defines the sample parameter *maskLayoutViewName*:

```
cdfParamId = cdfCreateParam(cdfDataId
?type           "cyclic"
?name           "maskLayoutViewName"
?prompt         "Type of layout"
?choices        ("layout" "layout.pwr")
?defValue       "layout"
?callback       "mosLayout( )")
```

The following functions define the sample parameter *width* with a callback from the parameter *w*:

```
cdfParamId = cdfCreateParam(cdfDataId
?type                "string"
?parseAsNumber "yes"
?units               "lengthMetric"
?parseAsCEL          "yes"
?storeDefault        "no"
?name                "w"
?prompt              "width"
?defValue            7u
?display             "artParameterInToolDisplay('w)"
?editable            "cdfgData->maskLayoutViewName->..."
?callback            "pcMOSw( )")
```

The following SKILL functions define the parameters *bend* and *bendt*:

```
cdfParamId = cdfCreateParam(cdfDataId
?type                boolean
?name                "bend"
?prompt              "PCell add bend?"
?defValue            nil
?display              nil)
```

```
cdfParamId = cdfCreateParam(cdfDataId
?type                float
?name                "bendt"
?prompt              "PCell add bend?"
?defValue            0.0
?display              nil)
```

The following SKILL function defines the sample parameter *diffD*:

```
cdfParamId = cdfCreateParam(cdfDataId
?type                boolean
?name                "diffD"
?prompt              "Drain pin diffusion only?"
?defValue            t
?display              "(cdfgData->bend->value) && ...")
```

Passing Parameters in a Design

In algebraic expressions that define the value of parameters, the *pPar*, *iPar*, *atPar,* and *dotPar* functions inherit the value of a parameter from a particular instance of the component.

- *pPar* means parent parameter, NLP (netlist processor) syntax [+*name*]
- *iPar* stands for instance parameter, NLP syntax [~*name*]
- *atPar* is named after the NLP syntax [@*name*]
- *dotPar* is named after the NLP syntax [.*name*]

The syntax of all four functions is:

```
_Par("parameter")
```

You can use these inherited parameter value functions to pass parameters in a design. If you change the value of one parameter, you also change every other parameter whose value is defined by an expression that includes a *Par* function of the parameter that you changed. Different *Par* functions can be used in the same expression.

You can define variables by expressions that use any combination of *pPar, iPar, atPar,* and *dotPar*. Those functions can reference other variables that are defined by expressions that include one of the *Par* functions. This chain can continue indefinitely. Circular references generate errors.

pPar

pPar("x") lets you reference the value of parameter *x* from the parent instance of the component.

An example of *pPar* used in a parameter definition is

```
rate = pPar("slewRate") + 100.0
```

When Analog Expression Language (AEL) evaluates this expression, it searches the parent instance for the value of `slewRate` and substitutes it into the expression. If it cannot find `slewRate` on the parent instance, AEL searches the parent cell's effective CDF for a value for `slewRate`.

Top-level schematic instances with parameters defined by expressions using *pPar* follow a special procedure. The netlister first searches the property list of the top-level schematic for a value for the referenced parameter. If it cannot find it, it searches the top-level cell's effective cell CDF for a default value for the parameter. This value can also be defined with an expression, but that expression can only include the *iPar* and *dotPar* functions.

iPar

iPar("x") lets you reference the value of a parameter on the current instance.

Suppose the parameter *i12* of a component is defined as a function of its input noise current, *n*:

```
i12 = V1/r12 + iPar("n")
```

The AEL evaluates this expression by searching the current instance for the value of *n*, which it substitutes into the expression. If the current instance does not have the parameter *n*, AEL uses the cell's effective CDF value as a default value for *n*. Changing the value of *n* does not have an effect unless you change the value of *n* on this specific instance.

Caution

Use the iPar function when the value of a parameter depends on the value of another parameter on the current instance but ensure that these parameters are not dependant on each other. For example, if you use the `cap` ***symbol and specify the length as iPar("w") and width as iPar("l") you will get a segmentation fault error as this results in infinite recursion.***

atPar

The use of *atPar* is discouraged. Use *pPar* instead. *atPar* searches the entire design hierarchy, looking at the parent, grandparent, and great-grandparent of the instance, and so on, until it finds the original value.

If *atPar* cannot find the parameter value anywhere in the design hierarchy, it then searches the current instance master, which is typically the simulation primitive for the device in a library. If that fails, *atPar* checks the global block for your simulator (*nlpglobals*).

dotPar

The use of *dotPar* is discouraged. It is not supported in several netlist situations.

When trying to evaluate a parameter on a primitive at the bottom level of a design hierarchy, *dotPar* behaves like *iPar*. If a property on a component or functional block higher up in the design hierarchy uses *dotPar("x")* in its definition, *dotPar* goes down to the bottom of the design hierarchy, searching for *x* on the leaf instance or primitive currently being output to the netlist.

At the top level of a design, *dotPar* searches the top-level cellview rather than the instance.

Inherited Parameters in Callbacks

Do not use callbacks based on parameters whose values are defined by expressions that include inherited parameter functions. Changes to the inherited parameters do not trigger the callbacks. Use callbacks only to establish dependencies between parameters whose values are numeric constants.

For example, in a transistor with the length parameter *L*, *L* is defined by a callback. Normally, changes in the value of the width parameter *W* trigger the *L* callback. The expression for this is

```
cdfgData -> L -> value = cdfgData -> W -> value
```

In this example, *W* is defined by the following expression:

```
W = pPar("test_width")*10u
```

If you change the value of *test_width*, the callback that defines *L* is not triggered, and the value of *L* becomes out of date.

Parameterized Cells

Although parameterized cells (pcells) provide great design flexibility, they also have unique requirements for their CDF descriptions. These requirements include the following.

- The default value for CDF parameters that are directly mapped to *pcell* stretch parameters must be the same as the minimum value of the *pcell* parameter.

  If the CDF default value is larger than the minimum *pcell* stretch parameter value, when you place a *pcell* with a parameter value less than the CDF default value, the *pcell* shrinks to the minimum *pcell* stretch value.

- In layout, *pcells* do not understand string variables. All variables must be floating-point numbers if you are using stretch parameters.

- In a schematic, the system places *pcell* symbols according to string variables because it must understand expressions (using the Analog Expression Language). Variables are most likely to be strings to allow scaling units (nanometers, micrometers, and so on).

The second and third items might appear to conflict, but you can use one of the two methods to simultaneously support the requirements of layout and schematics.

- Create two CDF parameters. One is a string variable for the schematic instantiation of the *pcell* symbol. The other is a floating -point number variable for the *pcell* layout view. You must maintain both variables.

- Create a callback for the floating point parameter based on the value of the string parameter. The user enters the string value and both values are set.

Complex Pole Example

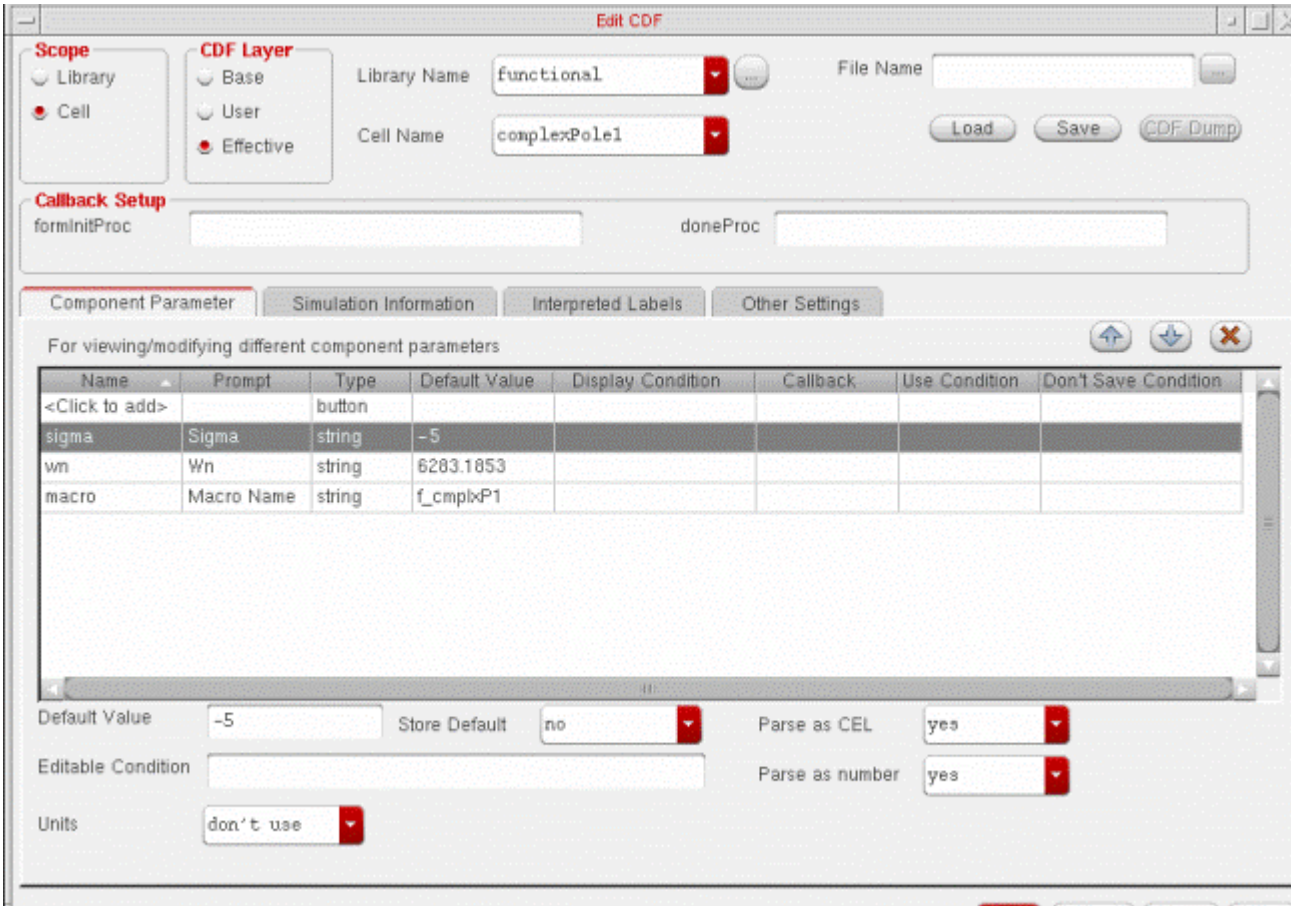
Using the same complex pole example from Chapter 2, “CDF Commands,” you should have the Edit CDF form open and selected the Component Parameter tab on the form.

The Component Parameter tab lists all component parameters. For *complexPole1,* these are *sigma*, *wn*, and *macro*.

Using the Component Parameter Tab

You can examine the different features of the Component Parameter tab using the *complexPole1* as an example.

➤ In the Component Parameter tab, select the *sigma* parameter.



The attributes for the parameter are displayed in the fields that appear in the bottom part of the Component Parameter tab.

The parameter *name* is set to *sigma*. The prompt used in the Edit Object Properties and Add Instance forms is *Sigma*. When you are done, you see the parameter name *Sigma*, but the system keeps track of the parameter by its real name, *sigma*. The parameter has a *Type* equal to *string*. The default value of *sigma* is -5, so every instance of the *complexPole1* in a schematic has this value initially. The *Parse as number* and *Parse as CEL* attributes are set to *yes* (true).

The first two parameters, *sigma* and *wn*, are component specific. The third parameter, *macro*, identifies the file that contains the macro (subcircuit) definition. The default value of *macro* is the *f_cmplxP1* file. Because *macro* is a CDF parameter, you can change its value to point to another file. This lets you create your own macros and reference them from the Edit CDF form. Editing at this level changes all instances of *complexPole1*. To change *complexPole1* in a particular design, use the *Edit Properties* command in the schematic editor.

If you want to make changes to the complex pole CDF description, copy the cell to a new library for which you have write access.

The complex pole example continues at the end of the next chapter, where you examine the options available for CDF simulation information.

NFET Example

This section shows you how to use the Edit CDF form to set up parameters and their attributes for the NFET cell. You should have the Edit CDF form open for the NFET cell created in Chapter 2, “CDF Commands.”

The sample NFET component requires a *maskLayoutViewName* parameter and five parameters needed for the *pcell* layout. The following table describes the parameters.

Parameter	Description
maskLayoutViewName	Cyclic field that distinguishes between two layout views
width	Floating-point number that defines the width of the gate
length	Floating-point number that defines the length of the gate
bend	Boolean to control conditional inclusion of shapes for bend
bendt	Boolean to help stretch shapes correctly when bend occurs

Parameter	Description
diffD	Boolean to control conditional inclusion of shapes for diffusion drain contact versus metal drain contact

Adding New Parameters

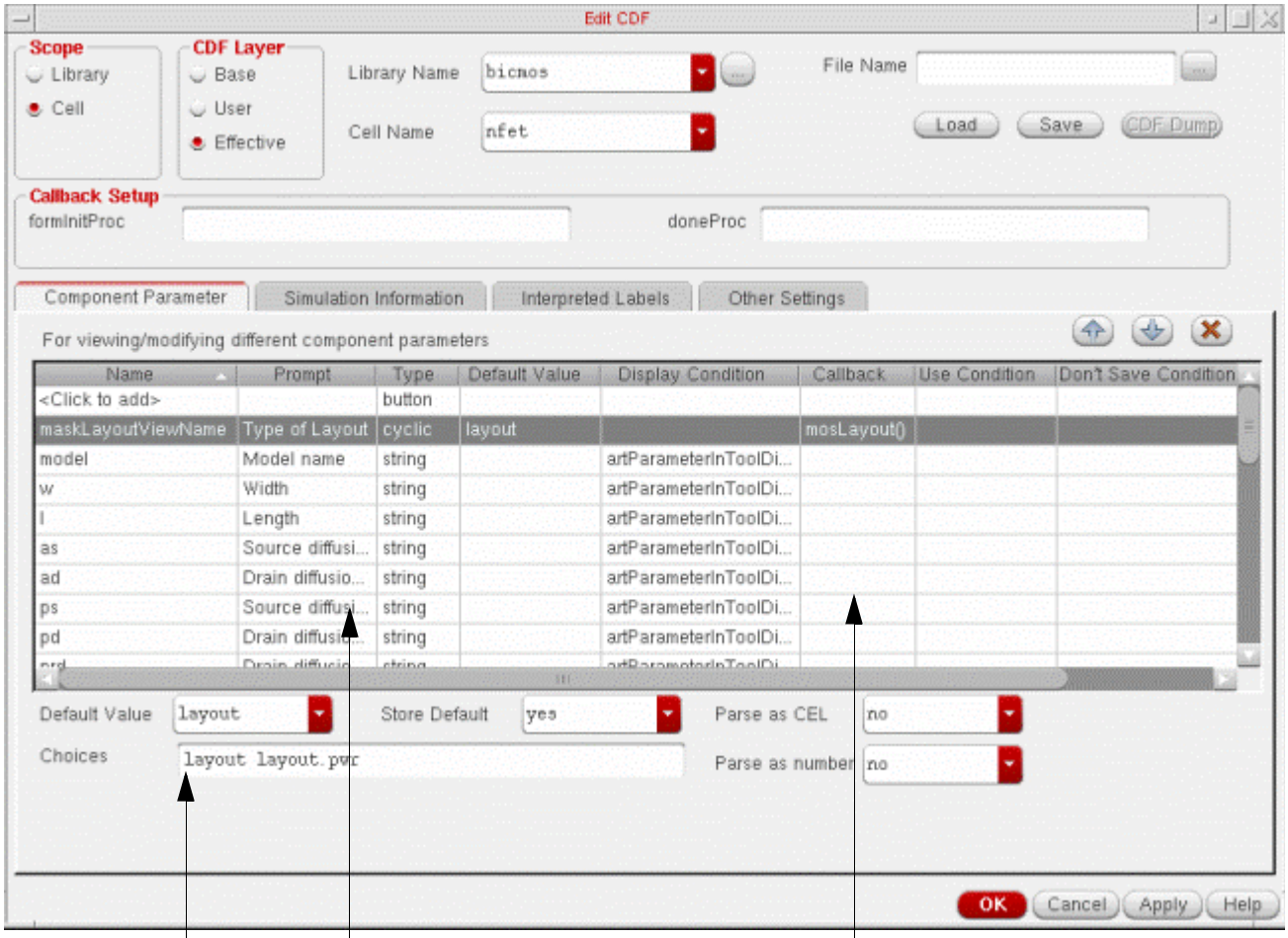
Use the following procedure to add new parameters to the NFET CDF description.

1. Click the Component Parameter tab on the Edit CDF form.
2. In the *Name* column, double-click where it says <*click to add*>, type `maskLayoutViewName`, then press the *Tab* key.

   The *maskLayoutViewName* parameter is displayed in a new row.

   maskLayoutViewName is a cyclic parameter that lets you choose between two layouts. The following figure describes decisions you must make about prompt use, parameter

type selection, and callback actions for the *maskLayoutViewName* parameter for the NFET component.



This cyclic field limits the user to two choices.

This prompt is more meaningful than the name required by Virtuoso Layout.

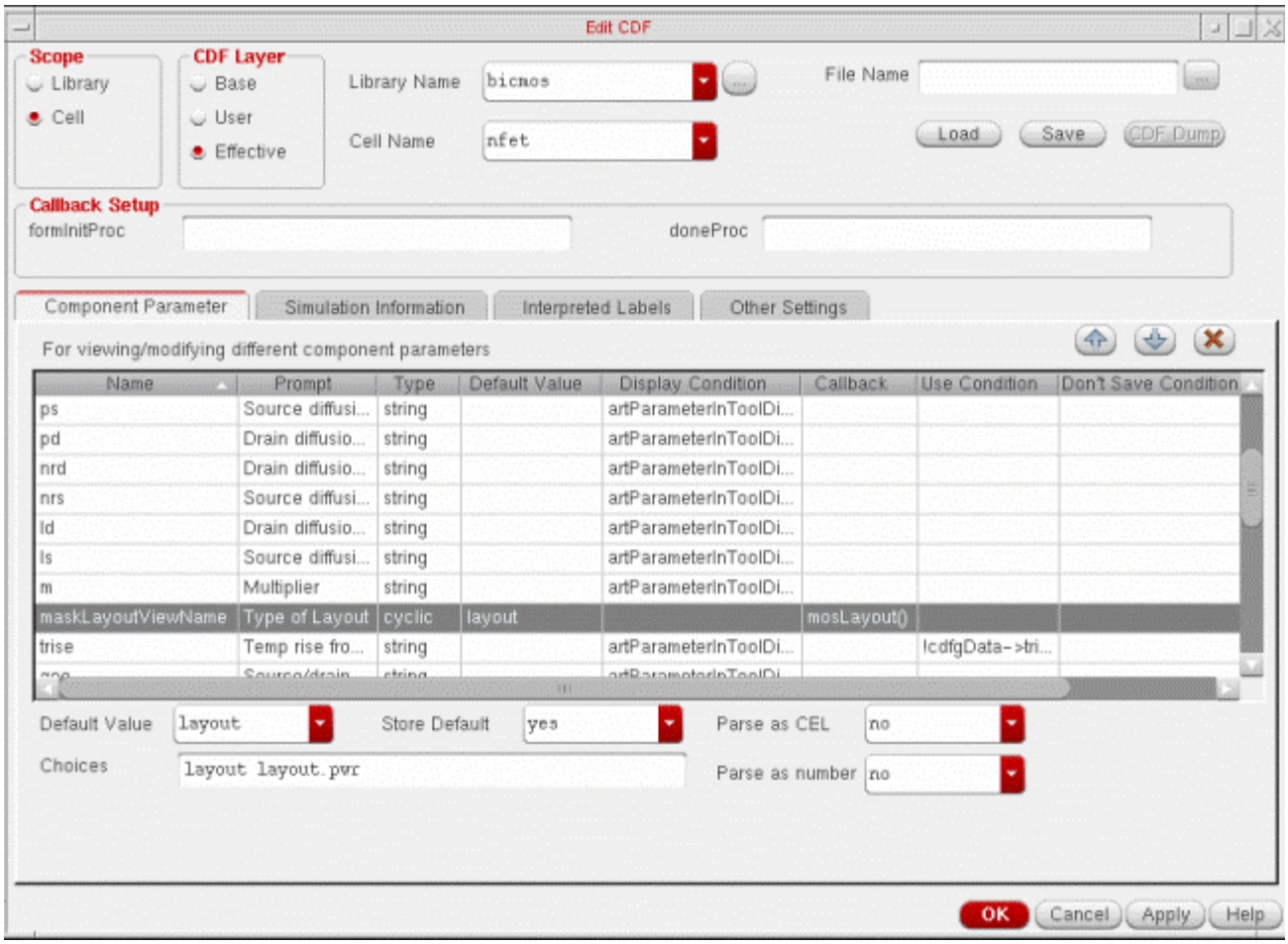
Use this callback to update width and length.

3. In the *Prompt* field, type `Type of Layout`, then press the *Tab* key.
4. In the *Type* cyclic field, select *cyclic*.
5. Double-click on the *Callback* field and type `mosLayout()`.
6. In the *Choices* field, type the following:

   ```
   layout layout.pwr
   ```

7. In the *Default Value* cyclic field in the bottom of the form, select *layout*.
8. In the *Store Default*, *Parse as CEL* and *Parse as number* cyclic fields, select *No*.

9. Click the button to move the `maskLayoutViewName` parameter after the parameter `m` as shown below:



10. When you are done, click *Apply* so that your entries take effect.

Setting Sample Callbacks

Because you copied the CDF description for the NFET component from the *nbsim* CDF description, there is already a width parameter *w* in the CDF setup for the simulators. You can set up a callback that updates the *width* parameter whenever the *w* parameter is updated. Then designers don't have to enter the width twice.

1. In the *Name* column, double-click where it says <*click to add*>, type `width`, then press the *Tab* key.

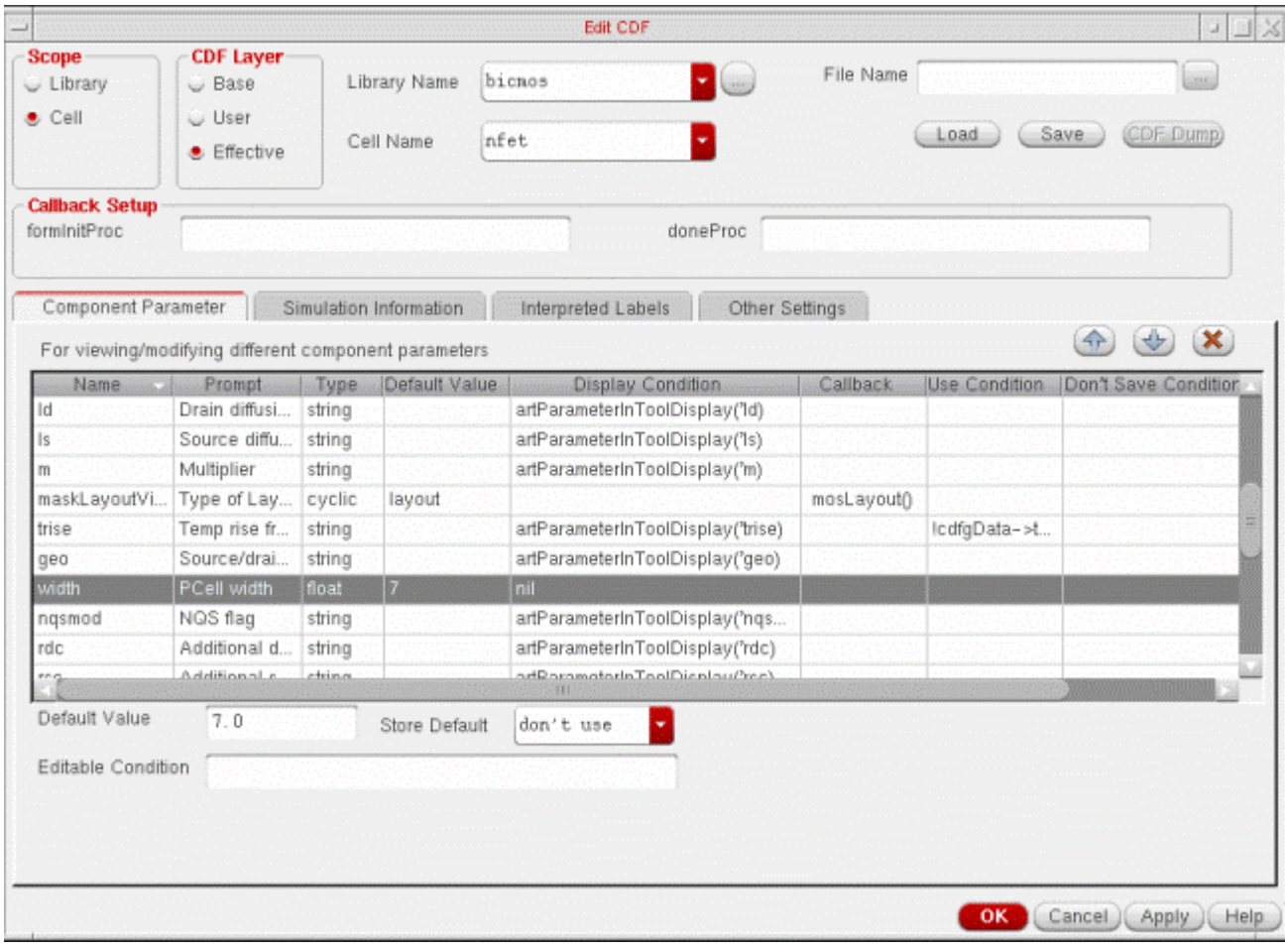
2. In the *Prompt* field, type `PCell width`, then press the *Tab* key.

3. In the *Type* cyclic field, select *float*.
4. In the *Default Value* field in the bottom of the form, enter a default value, say `7.0`, that matches the default value for the parameter *w*.
5. In the *Store Default* cyclic field, select *don't use*.

   The new width parameter does not need to be displayed in forms that display CDF information because the callback sets the value.
6. Double-click on the *Display Condition* field and type `nil`.

   This indicates that the `width` parameter will not be displayed in forms that display CDF parameters.

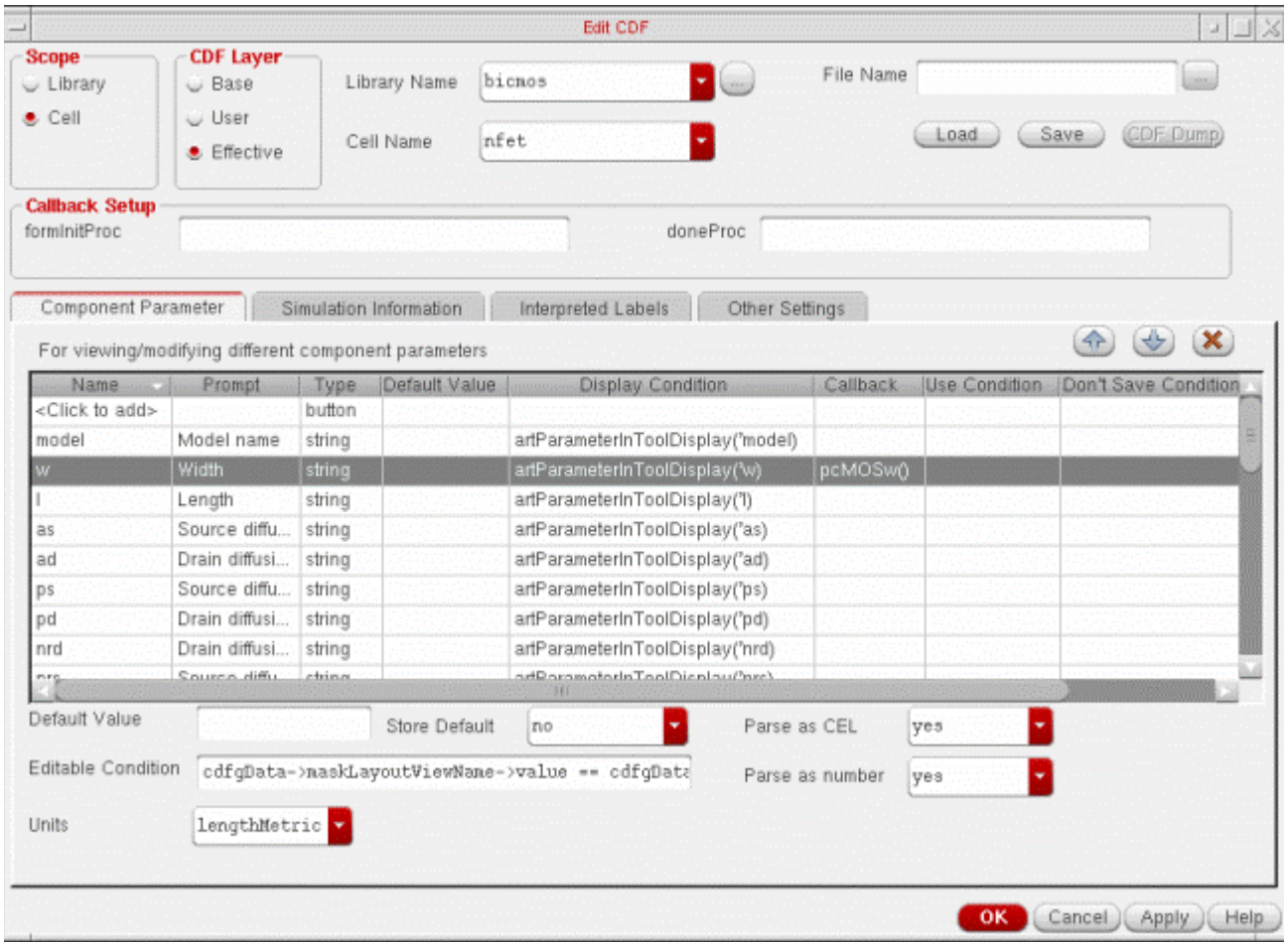
   Because this parameter is set automatically by another parameter, put it near the end of the parameter list.
7. Click the button to move the `width` parameter after the parameter `geo`.



8. Click *Apply* to save the new parameter.
9. In the Component Parameter tab, select the *w* parameter.
10. Double-click on the *Callback* field and type in the callback name `pcMOSw()`.

    (Refer to Chapter 7, "Writing Callbacks," for a description of callback procedures.)
11. In the *Editable Condition* field, make the parameter not editable by typing

```
cdfgData->maskLayoutViewName->value ==
    cdfgData->maskLayoutViewName->defValue
```



The expression in the *Editable Condition* field of parameter *w* controls the designer's ability to change this parameter. This expression evaluates to *nil* if the attribute *maskLayoutViewName* is not the same as its default value. In other words, if the designer selects the *layout.pwr* view name, this parameter is not editable.

The expression `artParameterInToolDisplay('w)` in the *Display Condition* field of parameter *w* (copied as part of the *w* parameter from the *analogLib* library) controls the

display of parameters when the user uses the *Edit – Properties – Tool Filter* command. The expression evaluates to *t* if this parameter is associated with the tool selected.

You can expand the *Display Condition* attribute expression. For example, to turn off the display of *w* whenever the *maskLayoutViewName* is *layout.pwr* (instead of making it uneditable), you can use the following:

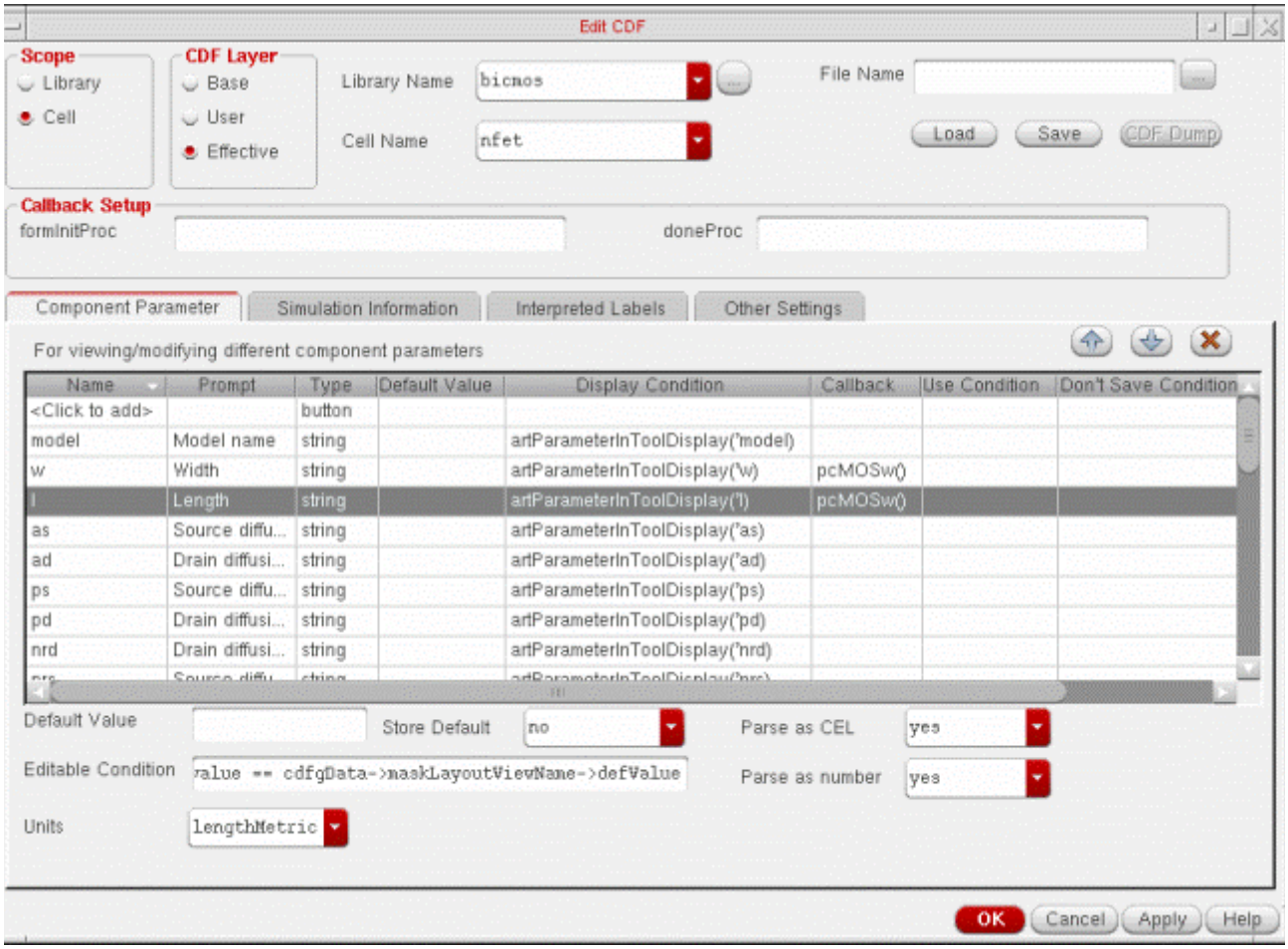
```
artParameterInToolDisplay('w) &&
    (cdfgData->maskLayoutViewName->value ==
    cdfgData->maskLayoutViewName->defValue)
```

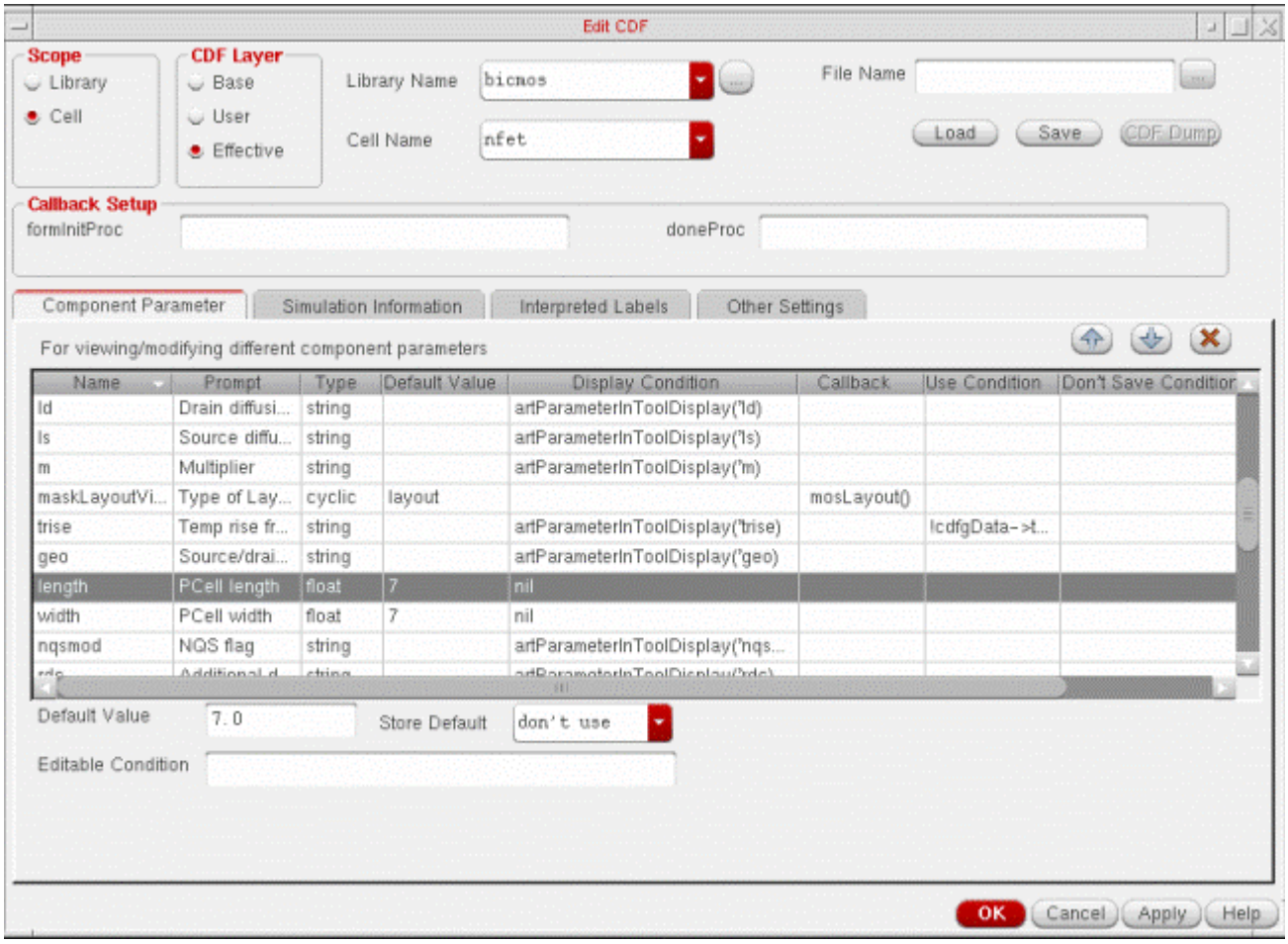
The function `artParameterInToolDisplay` takes a parameter name as its argument. It evaluates that parameter in the current CDF description and decides to display it or not, depending on whether the parameter is used by the simulator you are running.

Completing Parameter Definitions

Fill in the required information for the remaining parameters.

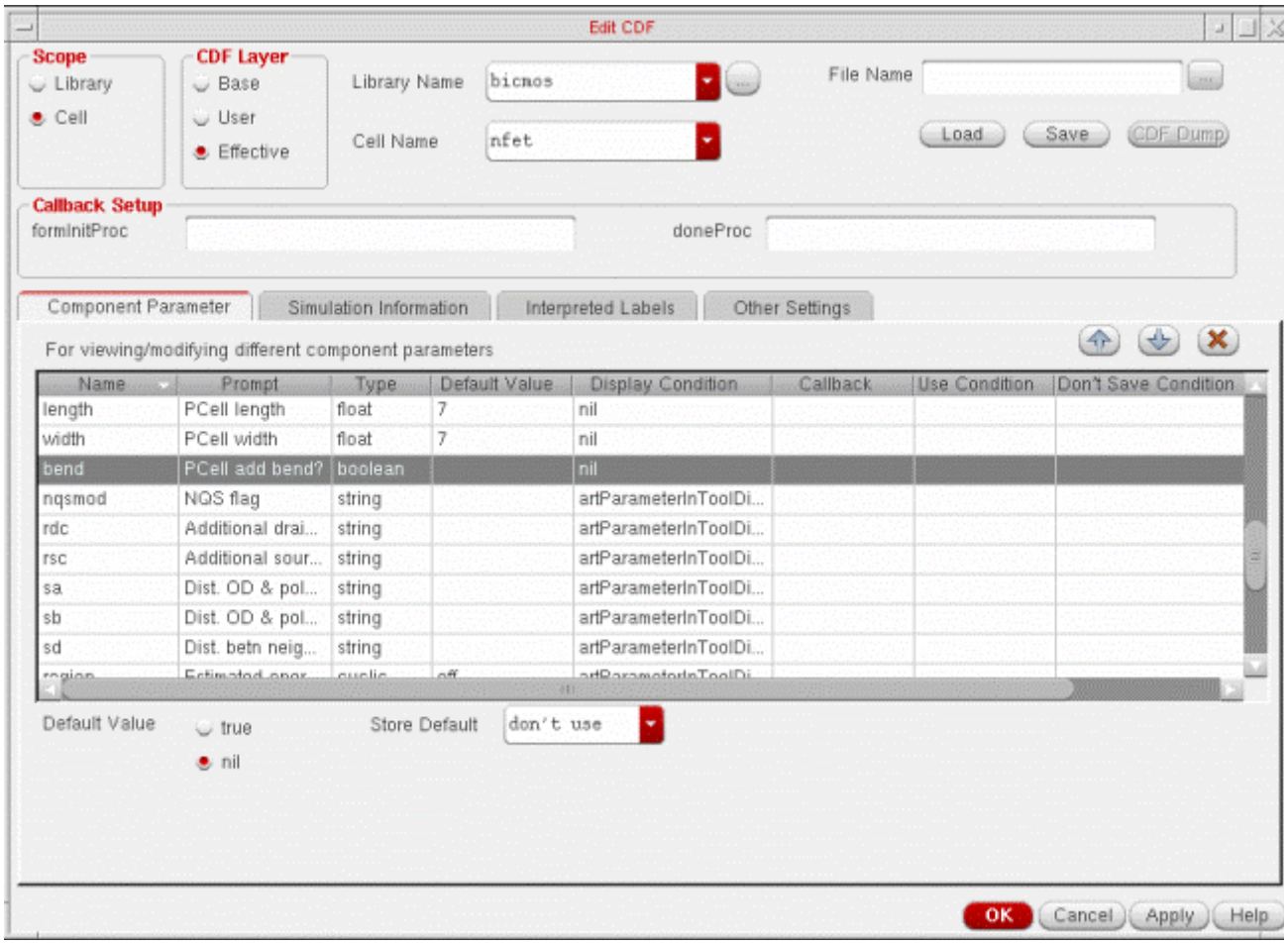
1. Repeat the previous procedure for the *l* and *length* parameters that you used for *w* and *width* as shown in the following figures.

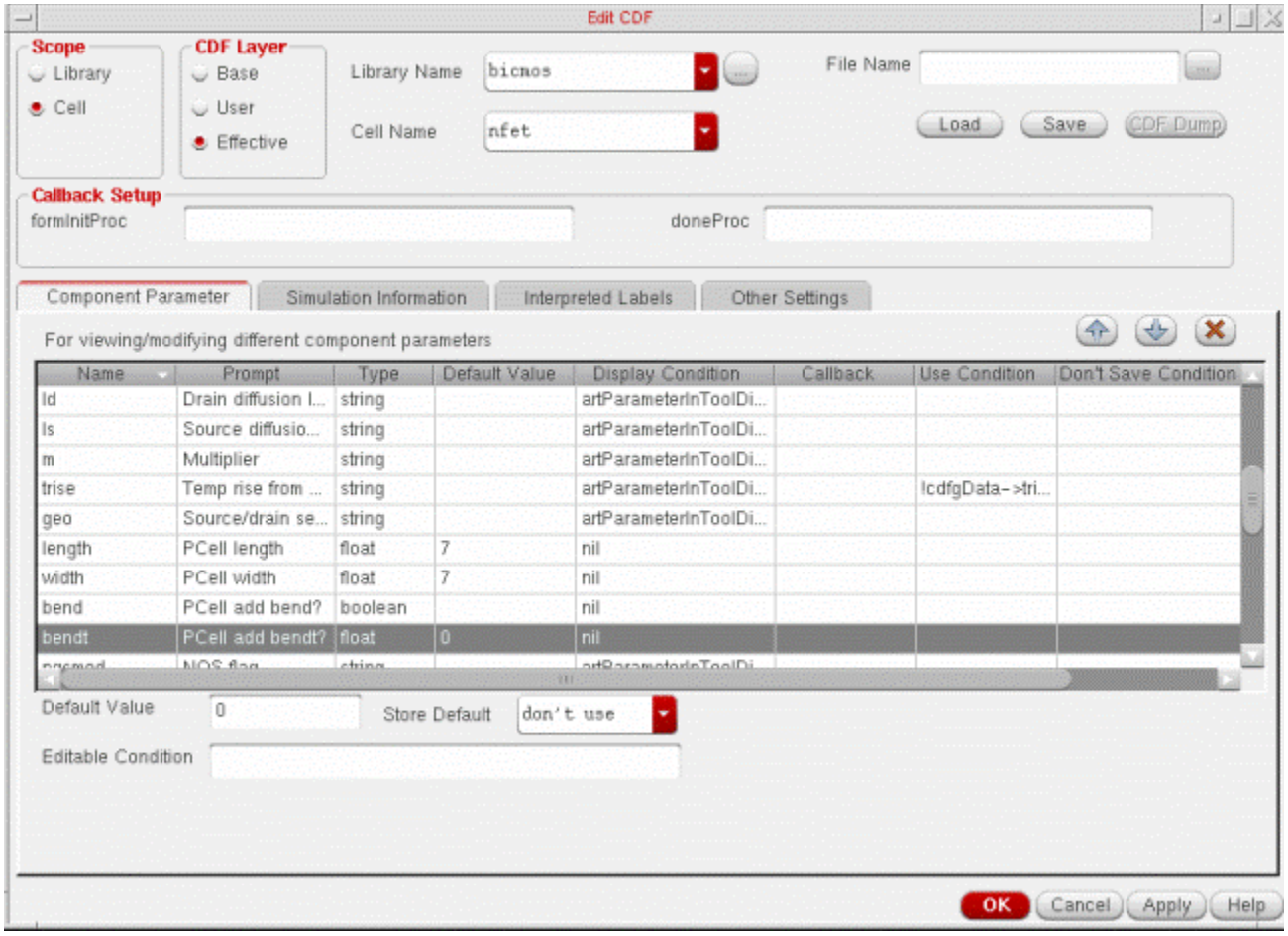




2. Using the Component Parameter tab again, add the parameters *bend* and *bendt* as illustrated in the following figure.

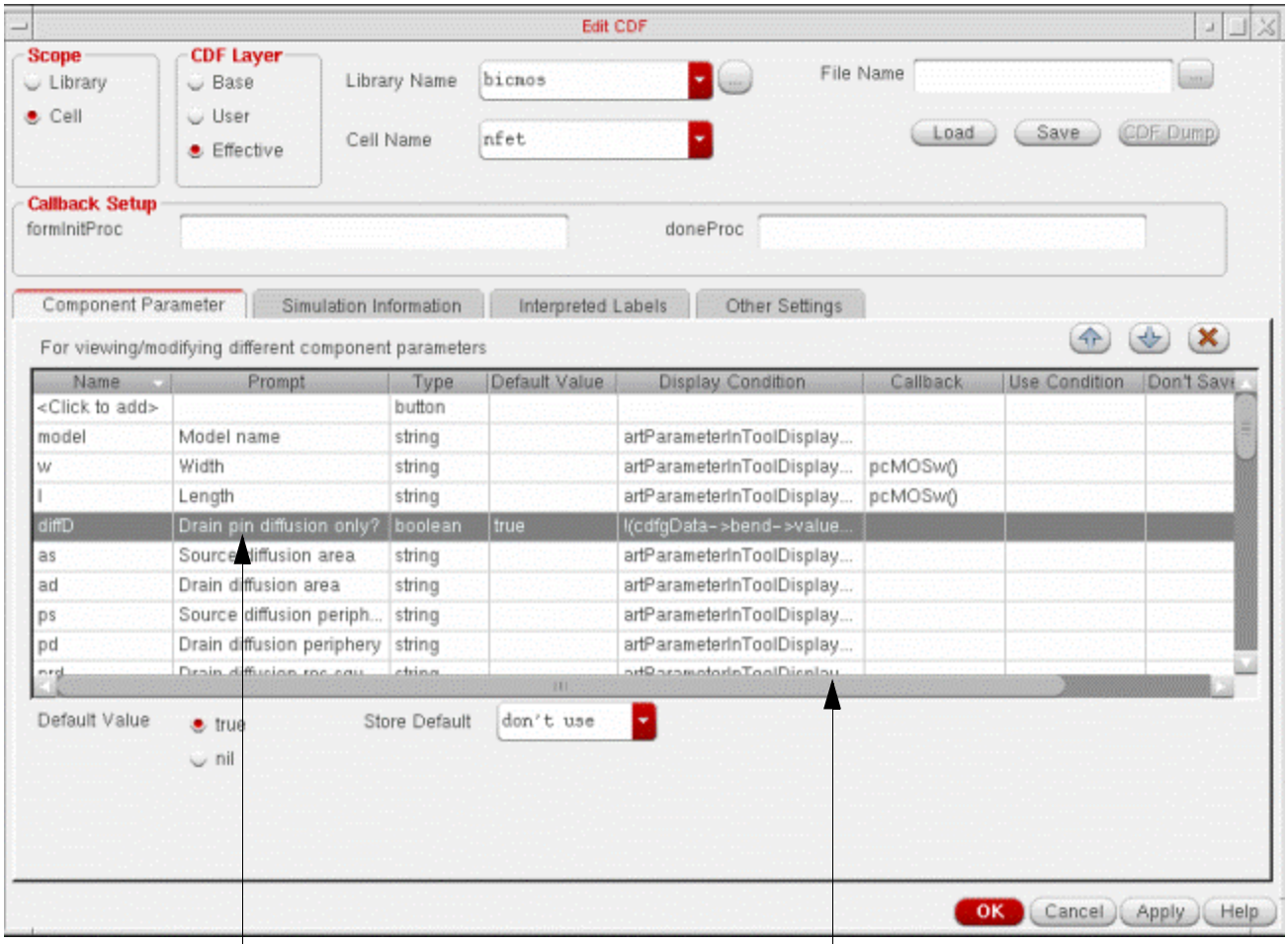
   The *pcell* parameters *bend* and *bendt* can also be controlled by the *width* parameter. You can place both parameters at the bottom of the parameter list because they are driven by the callback and, therefore, do not need to be displayed.


Edit CDF
Scope
Library
Cell
CDF Layer
Base
User
Effective
Library Name
bicmos
Cell Name
nfet
File Name
Load
Save
CDF Dump
Callback Setup
formInitProc
doneProc
Component Parameter
Simulation Information
Interpreted Labels
Other Settings
For viewing/modifying different component parameters
Name
Prompt
Type
Default Value
Display Condition
Callback
Use Condition
Don't Save Condition
length
PCell length
float
7
nil
width
PCell width
float
7
nil
bend
PCell add bend?
boolean
nil
nqsmod
NQS flag
string
artParameterInToolDi...
rdc
Additional drai...
string
artParameterInToolDi...
rsc
Additional sour...
string
artParameterInToolDi...
sa
Dist. OD & pol...
string
artParameterInToolDi...
sb
Dist. OD & pol...
string
artParameterInToolDi...
sd
Dist. betn neig...
string
artParameterInToolDi...
Default Value
true
nil
Store Default
don't use
OK
Cancel
Apply
Help



3. Add the *diffD* parameter as shown below.

The last *pcell* parameter is the boolean, *diffD*, which prompts the designer to specify whether or not to use diffusion to make contact with the drain. The display of this parameter depends on the value of *bend* and the type of layout selected. If you use *layout.pwr*, there is no reason to ask about the drain connection. The same is true if you are using a bend.



Place this parameter near the top of the parameter list because the user needs to select a value.

The display of this parameter depends on the value of bend and maskLayoutViewName parameters.

The complete expression in the *Display Condition* field is as follows:

```
!(cdfgData->bend->value) &&
    !(cdfgData->maskLayoutViewName->value=="layout.pwr")
```

You are ready to set the simulation options in the NFET CDF description, which are described in the next chapter.

4

Modifying Simulation Information

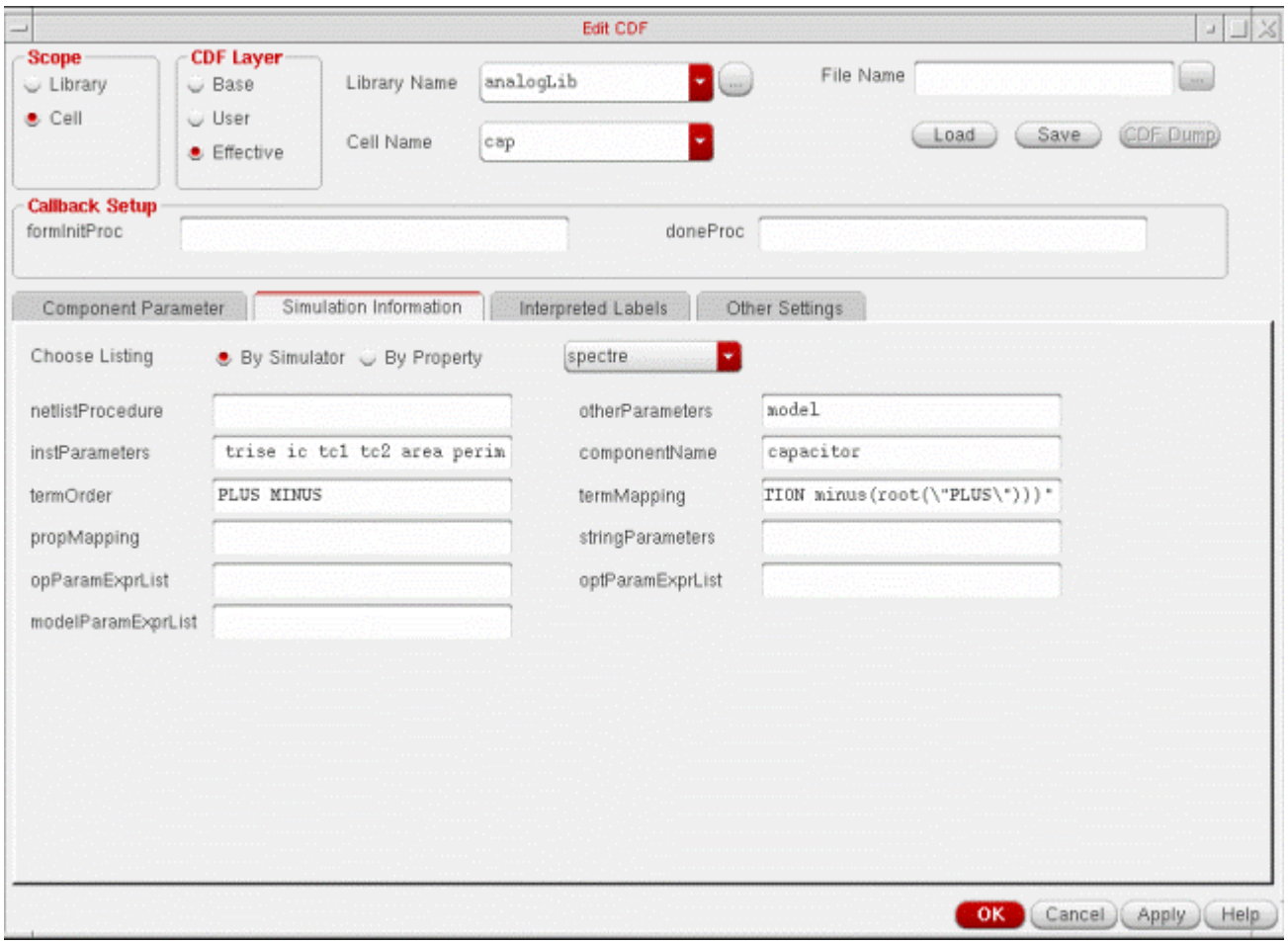
Overview

Before you can simulate a component, you must enter information required by each of the simulators you intend to use. The Simulation Information tab in the Edit CDF form lists the simulators associated with a CDF description. Use this tab to enter detailed information for each simulator.

Simulation Information Tab

Unlike the parameter attributes that you enter in the Component Parameter tab, the information you enter into the Simulation Information tab is a set of directions, parameters, and terminal names used by the simulator when it simulates the cell.

Each simulator requires a unique set of information. Not every field described in this tab appears for each simulator. Only those fields that are appropriate for the simulator you select are displayed. You can modify each of the fields in this tab.



Choose Listing

- **By Simulator** is a cyclic field that lets you select the simulator whose attributes you want to modify. When you select a new value, the form redisplays to show attributes for that simulator. After you change the attributes of one simulator, click on *Apply* before you change the attributes of another simulator.
- **By Property** is a cyclic field that lets you select the attribute whose values you to modify for various simulators. When you select a new value, the form redisplays to show the value of that attribute for all simulators. After you change the value of the attribute, click on *Apply* before you change the value of another attribute.

netlistProcedure is the name of the netlist procedure to use. Procedures that come with the system are defined in `<your_install_dir>/tools/dfII/etc/context`. You can also define your own procedures.

For the spectre interface, the netlister chooses a default netlist procedure if none is specified.

The netlist procedures for spectre that are supplied by Cadence start with spectre.

Note: The netlist procedure name can be defined as either a symbol (without quotation marks) or a string (with quotation marks). For example, *ansCdlHnlPrintInst* (symbol) or *"ansCdlHnlPrintInst"* (string).

enumParameters are parameters of SPICE and Spectre primitives, where the values of the parameters are enumerations. The parameter names must be names recognized by the targeted simulator.

For details see Appendix B, "Updating Legacy SimInfo for Analog Primitives" in the *Virtuoso AMS Environment User Guide*.

referenceParameters are parameters that have instance names as their values. The parameter names must be names recognized by the targeted simulator.

For details see the Appendix B, "Updating Legacy SimInfo for Analog Primitives" in the *Virtuoso AMS Environment User Guide*.

stringParameters are parameters that are treated as strings when they are written to the netlist. The parameter names must be names recognized by the targeted simulator.

For details see the Appendix B, "Updating Legacy SimInfo for Analog Primitives" in the *Virtuoso AMS Environment User Guide*.

arrayParameters are parameters that must be written to the netlist as arrays. The parameter names must be names recognized by the targeted simulator.

For details see the Appendix B, "Updating Legacy SimInfo for Analog Primitives" in the *Virtuoso AMS Environment User Guide*.

extraTerminals contains information for writing inherited connection terminals on instances. This information is used when the simulator view of an instance contains more terminals than are present on the symbol view.

For details see the Appendix B, "Updating Legacy SimInfo for Analog Primitives" in the *Virtuoso AMS Environment User Guide*.

otherParameters for spectre are parameters not defined in instParameters. These parameters are processed in a special way by the netlist procedure.

Typical values are *macro*, *model*, and *bn*. (The parameter names you list must reflect the property mapping you specify in the *propMapping* field.) If you use a tool filter, parameters that are not used by the tool are not displayed. Any field you specify in *otherParameters*, however, is displayed regardless of the tool filter (with the *artParameterInToolDisplay* function).

For details see the Appendix B, "Updating Legacy SimInfo for Analog Primitives" in the *Virtuoso AMS Environment User Guide*.

instParameters a list of simulator names of parameters that you want to include with this device in the netlist. (The parameter names you list must reflect the property mapping you specify in the *propMapping* field.)

For details see Appendix B, "Updating Legacy SimInfo for Analog Primitives" in the *Virtuoso AMS Environment User Guide*.

Some simulators require that a property have a different name or different capitalization from the parameter name in the parameter definition. For LVS (Layout Versus Schematic), this depends on the parameter names used in the LVS rules in your technology file. For example, the capacitor parameter might be *c,* and the LVS rules can refer to the property *C*. You can set the *propMapping* field to map the value of *c* to *C* so that the parameters listed with each instance are *C*.

componentName is the type of component you are creating, such as BJT, JFET, and MOSFET.

For the spectre interface, if the "model" parameter ("Model Name") has a value, that value is used as the name of the model in the netlist. Otherwise, the value of this entry is used as the name of the model. If neither the model parameter nor the *componentName* entry has a value, the name of the cell is used as the name of the model.

For details see Appendix B, "Updating Legacy SimInfo for Analog Primitives" in the *Virtuoso AMS Environment User Guide*.

When you are using *auLVS* and *auCDL*, component names can be anything. For other simulators, you must use lowercase component names similar to those used in *analogLib*. This is critical for SPICE types of netlists.

termOrder is a list of terminals that define the net order for your component. The nets are considered those nets connecting the terminals in the order given in this list.

For details see Appendix B, "Updating Legacy SimInfo for Analog Primitives" in the *Virtuoso AMS Environment User Guide*.

You can also use programmable nodes to define the net order. The use of programmable nodes is discouraged in releases 4.4.3 and later. Instead, use inherited connections. You can use the conversion tools to convert your designs. Programmable nodes are not supported by spectre.

To do this, you need to define a bulk node property (usually *bn*). The following entry is an example of *termOrder* input:

```
PLUS MINUS progn(bn)
```

This indicates that the third net, *progn*, is defined by the bulk node property instead of by a connection to a pin. The property *bn* must have a default value when using LVS.

Note: You can set a net name to a global net (such as *gnd!* or *vdd!*), to one of the instance pins (such as PLUS or MINUS), or to a net name, as long as the net, pin, or name exists in the cellview. You can assign a net name to *bn*. (This is called parameterizing the bulk node.) You can also set a net name to any other net. If the netlisting is flat, the net name must be fully qualified, identifying all levels of hierarchy in the design. If the netlisting is hierarchical, the net name does not need be fully qualified if it is a local net name on the same page.

Caution

You must escape invalid characters while specifying termOrder using symbols to make them valid symbols in SKILL. All symbols must be preceded by the backslash (\) as used in the following code:

```
cdfId = cdfCreateBaseCellCDF( cellId )
.
.
.
cdfId->simInfo->spectre = '(nil
    otherParameters (model)
    instParameters (w l as ad ps pd nrd nrs ld ls m trise region)
    termOrder ( a\<0\> a\<1\> )
    termMapping (nil D \:d G \:g S \:s B \:b )
    )
```

deviceTerminals lets you define the terminal names to be listed in the device definition line. If you leave this field blank, the terminals listed use the same names as those in the *termOrder* field. Used by *auLVS* only.

The entries in this field are usually different from those in the *termOrder* field when you have a programmable node (the bulk node). For example, if you used *progn(bn)* in the *termOrder* field for the three-terminal capacitor, you might see the following entry in *deviceTerminals*:

```
PLUS MINUS BULK
```

termMapping is used when the simulator has different names for the component terminals than the cell schematic. It is set up as a disembodied property list. Therefore, the first entry in the list is `nil`. This is followed by a set of name-value pairs, where the first name is the

terminal name on the symbol and the second name is its corresponding terminal name used by the simulator for the device.

For example, in the *nbsim* cell in *analogLib*, the terminals names *D*, *G*, and *S* are specified in uppercase. Although uppercase terminal names are appropriate for SPICE, the Spectre simulator expects the terminal names *d*, *g*, and *s* in lowercase. Therefore, in the SKILL version, to map between the two sets of terminal names for the Spectre simulator, you can enter the following code in the *simInfo* field:

```
termMapping(nil D \:d G \:g S \:s B \:b)
```

Note: Make sure all non-alphanumeric characters are preceded by a backslash (\).

This is important when you use backannotation because the simulator saves the results under the terminal name that it recognizes.

propMapping lets you enter your own CDF parameter names in place of the names recognized by the simulator. You can define different name mappings for each simulator. A single parameter can be identified a different way for each simulator.

For details see Appendix B, "Updating Legacy SimInfo for Analog Primitives" in the *Virtuoso AMS Environment User Guide*.

You specify an entry in the *propMapping* field in the following format:

```
nil simParamName1 CDFparamName1
    simParamName2 CDFparamName2 ...
```

This format represents a disembodied property list, so the first element is always *nil*.

The LVS *propMapping* list is *nil C c*. The CDF parameter name is *c*, but under the LVS rules, it must be *C*. In another simulator, the same CDF parameter value might be referred to as *F*. In this case, the *propMapping* list for this simulator is *nil F c*. A third simulator might refer to this value as *cap*, making its *propMapping* list *nil cap c*.

modelName is used only by the analog-microwave version of CDL.

permuteRule, used only in *auLVS*, lets you indicate the permutability of the pins, that is, whether pins can be swapped or if they must be fixed. For example, the capacitor in the *propMapping* definition shows that the PLUS and MINUS terminals are permutable.

namePrefix lets you enter a prefix for defining the instance in the netlist. This is not used by the spectre interface. Typically, this is the first letter of your component. The netlister takes this prefix (R, for example), and the name of your component (12, for example), and forms the name (R12).

You can also use a function or expression to evaluate the prefix. Ensure that the function or expression returns a string value.

- To use a function to evaluate the prefix, specify the prefix as:

  ```
  FUNCTION myFunction
  ```

  where `myFunction` is the name of the function that returns the prefix value. The function must be defined in your `.cdsinit` or `.cdsenv` file.

- To use an expression to evaluate the prefix, specify the prefix as:

  ```
  FUNCTION myExpression
  ```

  where `myExpression` is any expression that returns a prefix value. For example, specify the following expression to use the value of the `myPrefix` property on an instance as the prefix:

  ```
  FUNCTION cdfGetInstCDF(hnlCurrentInst)->myPrefix->value
  ```

current lets you save the simulation and definition of the current plot set. This is not used by the spectre interface. The only choices for this field are *port* or *nil*. If you do not specify a choice, no currents are saved. You can choose to save or plot the current for the port of a device or through a component. For SPICE-based simulators, you cannot save currents on specific ports. Currents can be saved only on the component (for example, resistors). The Spectre simulator lets you save currents on specific ports. For devices (for example, an *npn* transistor), you probably want to save currents on a port.

Typically, only the positive port current of a component current is written to the simulation PSF file, and the Cadence Analog Design Environment software calculates the negative port by taking the negative of the positive port. To use this feature to extract component currents, you must set the *termMapping* field as follows:

```
termMapping =
    (nil PLUS \:p MINUS "(FUNCTION minus(root(\"PLUS\")))")
```

modelParameters is used only by the microwave simulator HP MNS.

opParamExprList lets you specify DC operating point information.

optParamExprList lets you specify transient operating point information.

modelParamExprList lets you specify model parameters.

Tip

For more information and examples on opParamExprList, optParamExprList, and modelParamExprList, see Accessing Subcircuit Model Parameter and Operating Point Information on page 204.

Simulator Options

The following table lists six simulators and the options that apply to them.

Option	spectre	auLVS	ams	auCdl	UltraSim	hspiceD
netlistProcedure	Yes	Yes	Yes	Yes	Yes	Yes
otherParameters	Yes	Yes	Yes	Yes	Yes	Yes
instParameters	Yes	Yes	Yes	Yes	Yes	Yes
enumParameters	No	No	Yes	No	No	No
modelArguments	No	No	No	No	No	No
macroArguments	No	No	No	No	No	No
componentName	Yes	Yes	Yes	Yes	Yes	Yes
termOrder	Yes	Yes	Yes	Yes	Yes	Yes
deviceTerminals	No	Yes	No	No	No	No
termMapping	Yes	No	Yes	No	Yes	Yes
propMapping	Yes	Yes	Yes	Yes	Yes	Yes
excludeParameters	No	No	Yes	No	No	No
extraTerminals	No	No	Yes	No	No	No
isPrimitive	No	No	Yes	No	No	No
permuteRule	No	Yes	No	No	No	No
namePrefix	No	Yes	No	Yes	Yes	Yes
current	No	No	No	No	No	No
arrayParameters	No	No	Yes	No	No	No
stringParameters	Yes	No	Yes	No	No	No
referenceParameters	No	No	Yes	No	No	No
opParamExprList	Yes	No	Yes	No	No	No
optParamExprList	Yes	No	Yes	No	Yes	Yes
modelParamExprList	Yes	No	No	No	Yes	No
dollarParams	No	No	No	Yes	No	No

Option	spectre	auLVS	ams	auCdl	UltraSim	hspiceD
dollarEqualParams	No	No	No	Yes	No	No
modelName	No	No	No	Yes	No	No

Editing Simulator Options

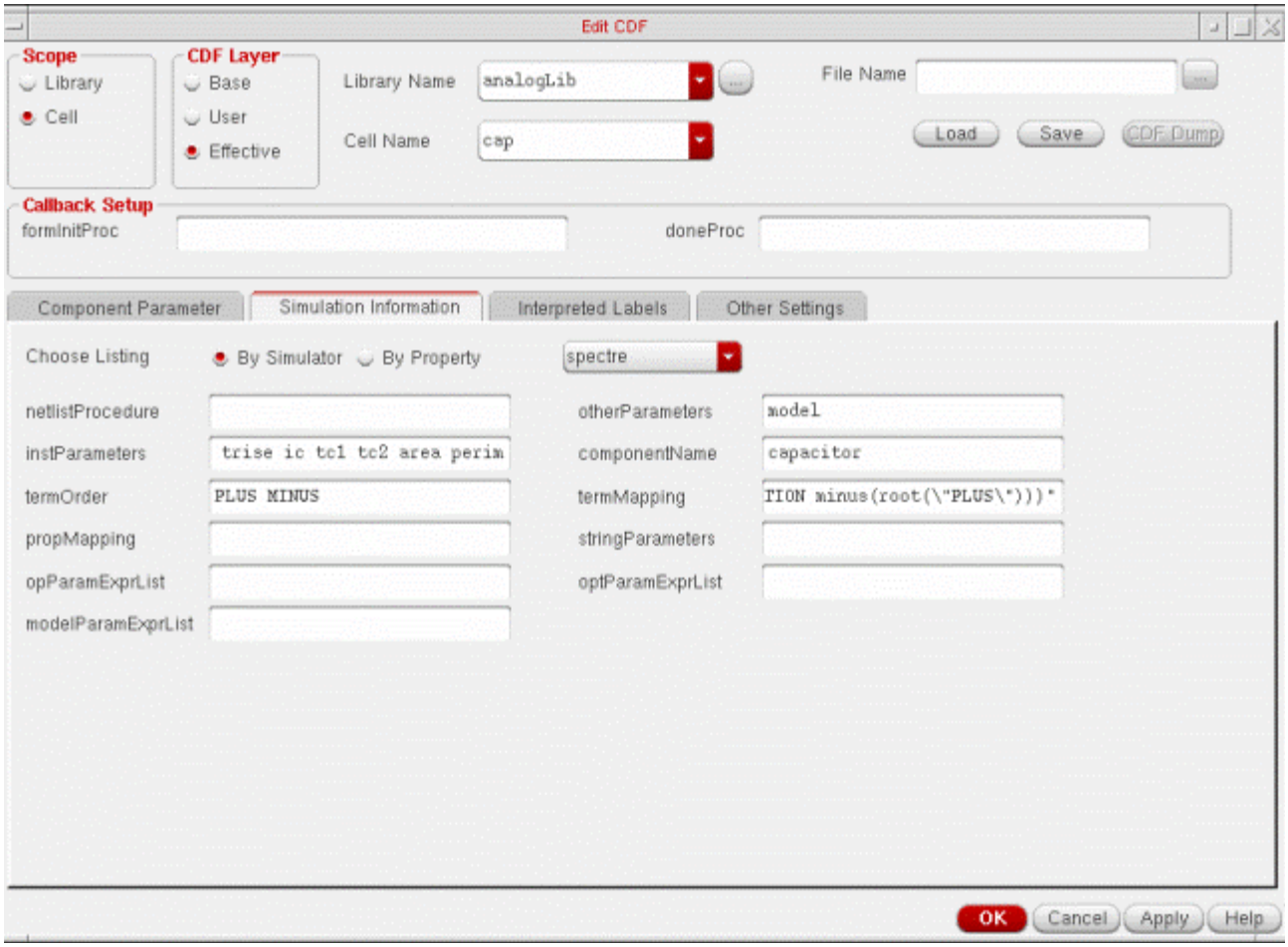
Setting simulator options can be a lengthy process. You should copy CDF simulator information from other components and libraries whenever possible. This section describes editing simulator options with the Edit CDF form and with SKILL functions.

Using the Edit CDF Form

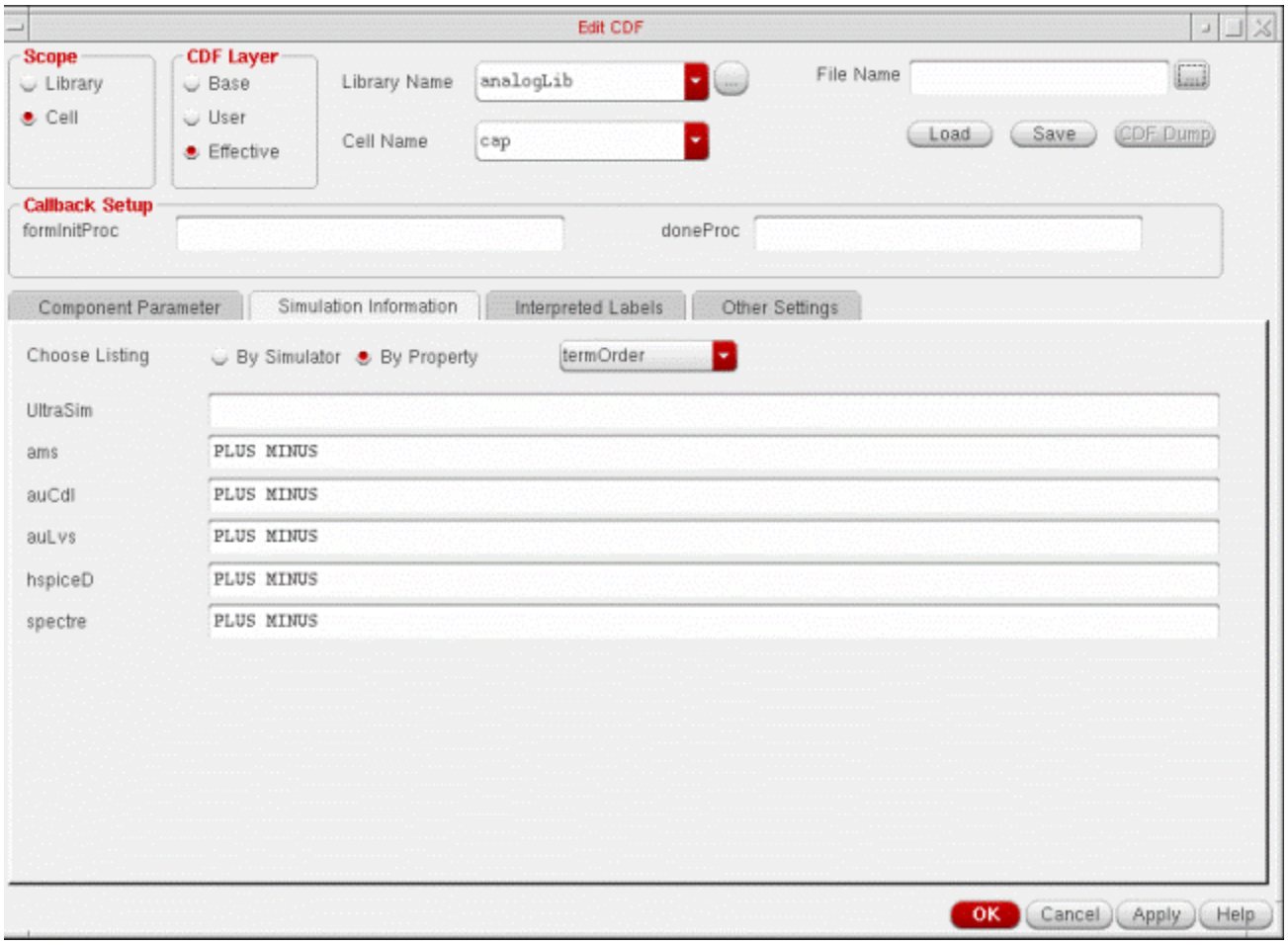
Use the following procedure to set simulator options with the Edit CDF form:

1. Click the Simulation Information tab in the Edit CDF form.
2. Do one of the following:

- To specify the settings for a simulator, select the *By Simulator* radio button and choose the simulator in the *Choose Listing* cyclic field. Add or modify the simulator settings as needed.



- To specify the settings for a property, select the *By Property* radio button and choose the property in the *Choose Listing* cyclic field. Add or modify the settings for the property for each simulator as needed.



3. Click *Apply.*

4. Click the Component Parameter tab to check that all simulator parameters are defined.

Note: When you edit the simulator information the CDF termOrder may not be automatically updated. Instead, you may get the *Want to Update* dialog box to confirm overwriting the existing CDF termOrder.
You can suppress the Want to Update dialog box by setting the .cdsenv variable `queryCDFtermOrder` to `nil`. This will ensure that the CDF is automatically updated without checking for confirmation. You can undo the automatic update by setting this variable back to its default value, `t`.

Using SKILL

You can use the SKILL variable *cdfId* and the right arrow (->) operator to specify simulator options. Enter the options for each simulator as elements of a disembodied property list.

The following SKILL expression sets the options for the Spectre (spectre) simulator:

```
cdfId = cdfCreateBaseCellCDF( cellId )
.
.
.
cdfId->simInfo->spectre = '(nil
    otherParameters (model)
    instParameters (w l as ad ps pd nrd nrs ld ls m trise region)
    termOrder ( D G S B )
    termMapping (nil D \:d G \:g S \:s B \:b )
    )
```

The following SKILL expression sets the options for the Analog LVS (auLvs) simulator, given a definition for *cellId*:

```
cdfId = cdfCreateBaseCellCDF( cellId )
.
.
.
cdfId->simInfo->auLvs = '(nil
    netlistProcedure ansLvsCompPrim
    instParameters (m l v)
    componentName n0bsim
    termOrder (D G S progn(bn))
    deviceTerminals (D G S B)
    propMapping (nil L l W w)
    permuteRule "(p D S)"
    namePrefix "Q")
```

The following SKILL expression sets the options for the Spectre simulator:

```
cdfId = cdfCreateBaseCellCDF( cellId )
.
.
.
cdfId->simInfo->spectre = '(nil
    netlistProcedure ansSpectreDevPrim
    otherParameters (bn model)
    instParameters (w l as ad ps pd nrd nrs ld ls m trise)
    componentName bsim1
    namePrefix "S"
    current port)
```

The following SKILL expression sets the options for the Cadence SPICE (cdsSpice) simulator:

```
cdfId = cdfCreateBaseCellCDF( cellId )
.
.
.
cdfId->simInfo->cdsSpice = '(nil
    netlistProcedure ansSpiceDevPrim
```

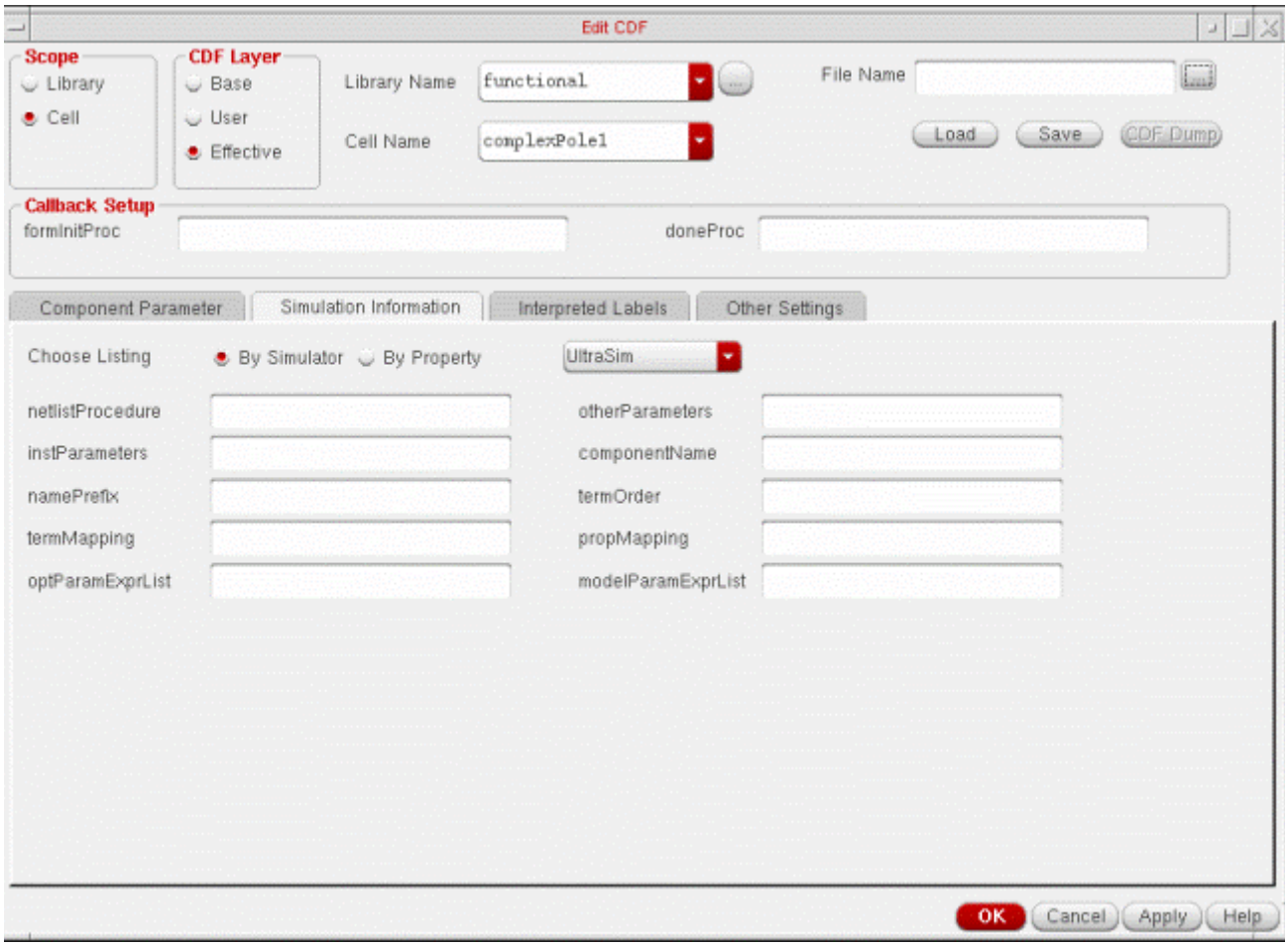
```
otherParameters (model bn)
instParameters (m w l ad as pd ps nrd
    nrs ld ls off vds vgs vbs)
componentName bsim
termOrder (D G S progn(bn))
propmapping (nil vds Vds vgs Vgs vbs Vbs)
namePrefix "S"
current port)
```

Complex Pole Example

In this portion of the complex pole example, you use the Simulation Information tab to look at the simulation information provided with the *complexPole1* component.

1. Click the Simulation Information tab on the Edit CDF form.

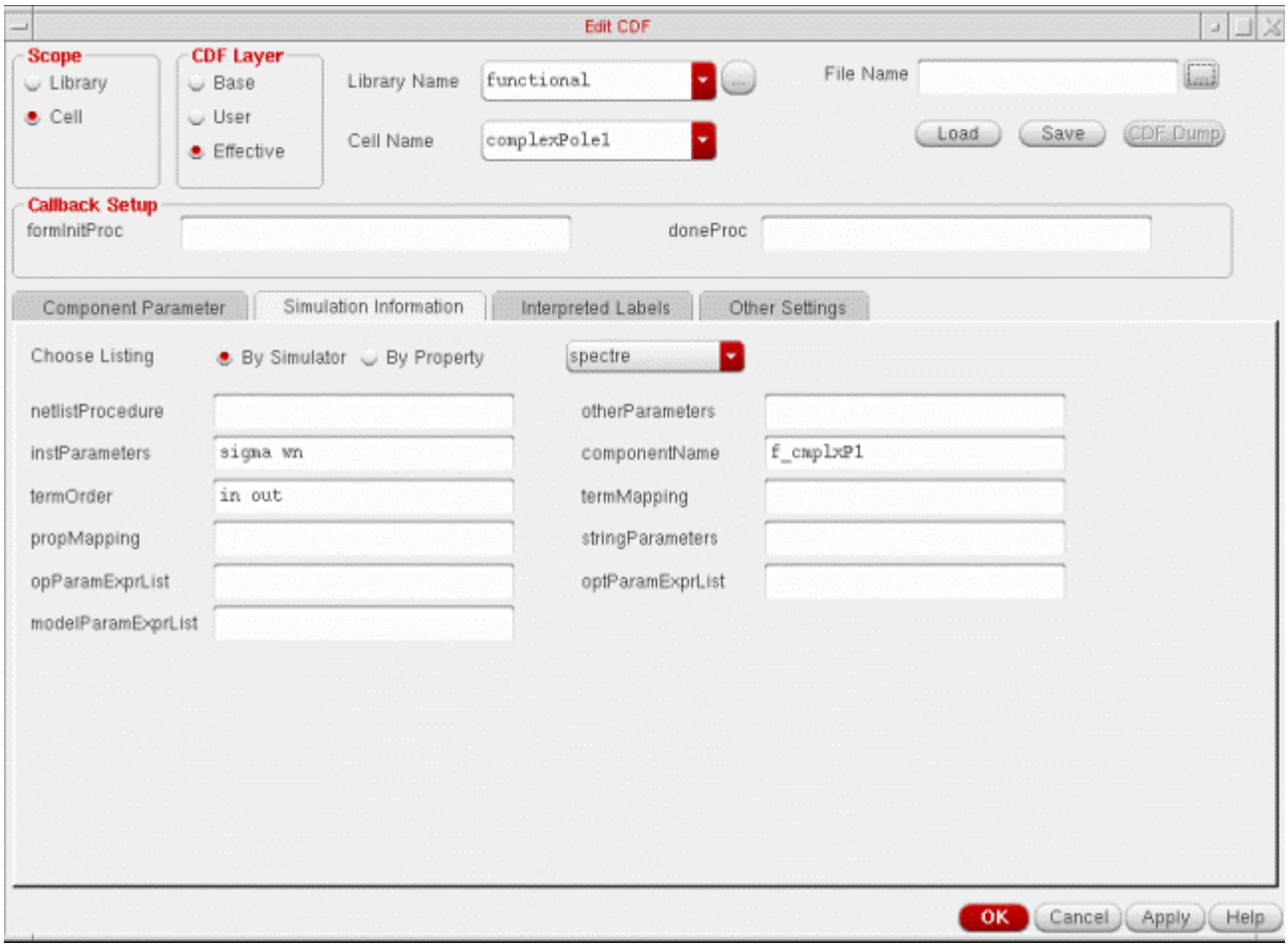
   This tab shows the simulator-specific data.



2. Select the *By Simulator* radio button and choose a simulator in the *Choose Listing* cyclic field. See the details for each simulator.

3. In the *Choose Listing* cyclic field, select *spectre*.

   The form displays all of the fields applicable to *spectre*. However, not all fields are required by different netlist procedures.



For *spectre*, *complexPole1* requires no netlist procedure.

The other important fields are *termOrder* and *instParameters*. The *termOrder* field is an ordered list of the terminals as named on the symbol. The netlister uses this list. The order is important, especially if the component is defined by a subcircuit: the order in *termOrder* maps to the node numbers specified in the first line of the subcircuit.

Because *complexPole1* uses a macro model (subcircuit definition) to define its operation, the CDF parameter model must have a value in the Component Parameters section for

spectre. If you are using a schematic to describe the component, the model information is unnecessary.

For the spectre interface, the *instParameters* field determines the names of the parameters of all types of components, primitives, subcircuits, model cards, and Verilog-A modules. The subcircuit definition for f_cmplxP1 is

```
subckt f_cmplxP1 (1 2)
parameters sigma wn
E1 3 0 1 0 1E-6 (??)
R1 (4 5) resistor r=1
L1 (3 4) inductor l=(1/(-2*sigma))
C1 (5 0) capacitor c=(-2*(sigma))/((wn*wn)+(sigma*sigma))
ends
```

This model, and others used by the functional block library, are in the following file:

your_install_dir/tools/dfII/etc/cdslib/artist/spectre/functional.scs

The order of parameters is particularly important in the case of macro models. The first parameter in the list is assigned to the variable *&2* in the subcircuit description, the second to *&3*, and so forth. In this example, *complexPole1* is defined by a macro model and the parameters *sigma* and *wn* on the symbol instance are passed to the model file *f_cmplxP1* as *&2* and *&3*.

The model file (*f_cmplxP1*) for the complex pole is

```
.SUBCKT &1 1 2
E1 3 0 1 0 1E-6
R1 4 5 1
L1 3 4 (1/(-2*(&2)))
C1 5 0 (-2*((&2)))/(((&3)*(&3))+((&2)*(&2)))
.ENDS &1
```

This model, and others used by the functional block library, are in the following directory:

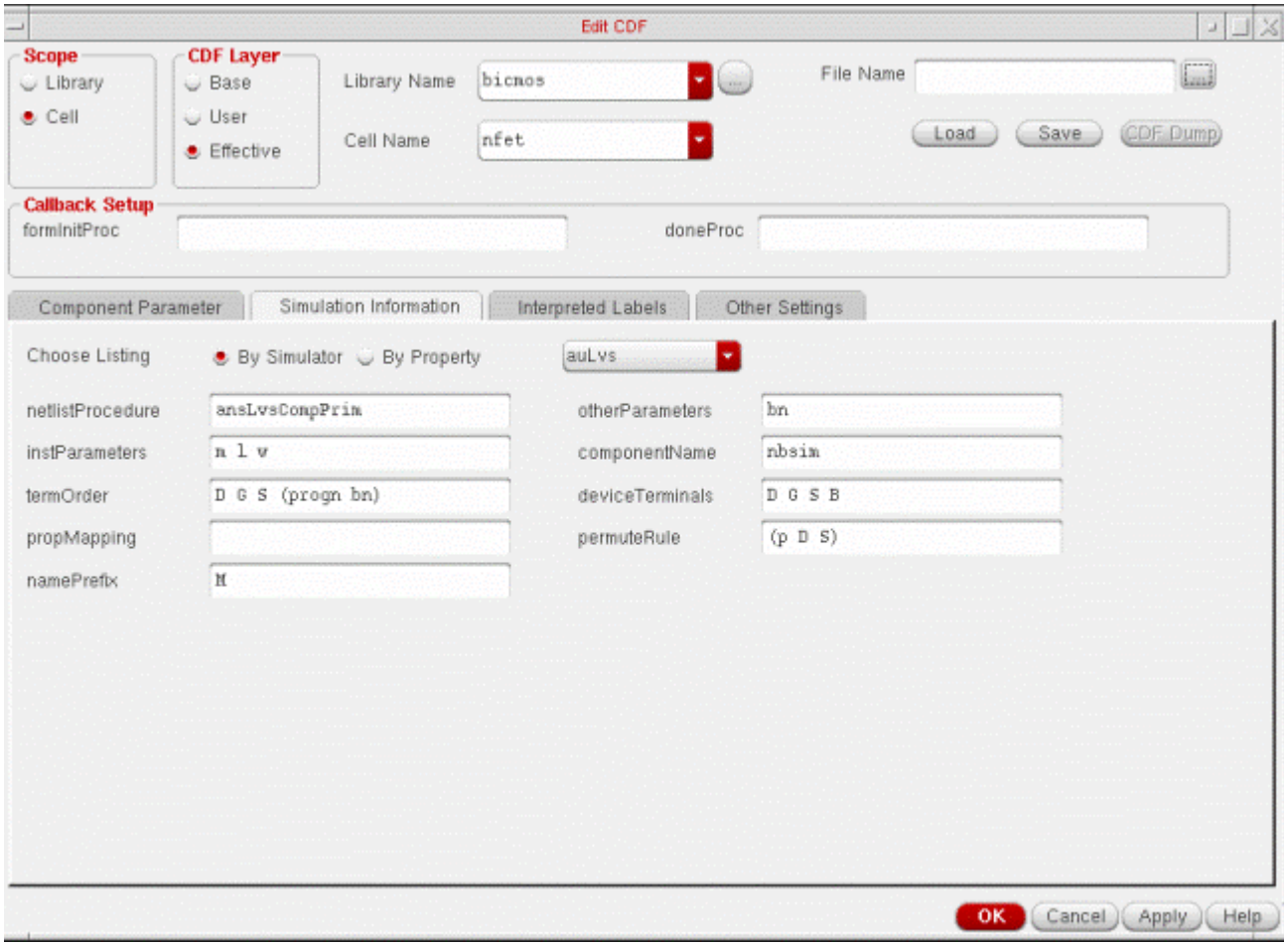
your_install_dir/tools/dfII/etc/cdslib/artist/cdsSpice/functional

NFET Example

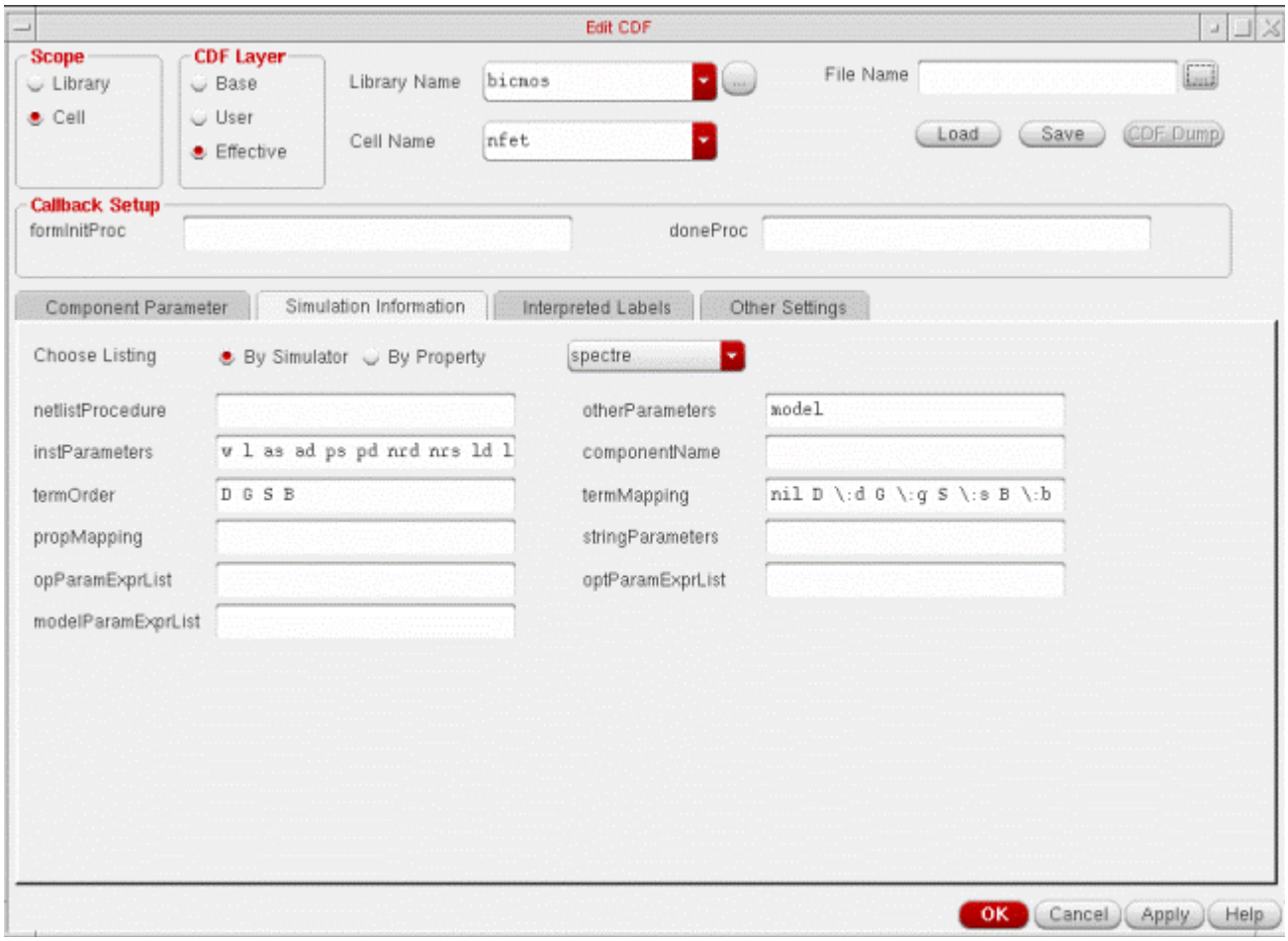
Before proceeding with this example, open the Edit CDF form and set it to edit the *nfet* cell in the *bicmos* library. Click the Simulation Information tab.

For this NFET example, the next step in CDF development is to select simulator options for the new component. Because you copied the CDF description for the NFET component from *analogLib*, there are five simulators with their options already defined, including *auLvs* and *spectre*.

For the Analog-Microwave LVS (auLvs) simulator, you need to define three parameters in the parameter section of the CDF description, *m*, *l*, and *w*. For the NFET example, these parameters were copied from the *analogLib nbsim* cell.



You also need parameters for the spectre interface but these are already defined in the original CDF that was copied. The Spectre simulator does not require property mapping.



5

Specifying Label Information

Interpreted Labels Information

You can use interpreted labels to display parameter values, evaluated parameter values, net connectivity information, backannotated simulation information, and more.

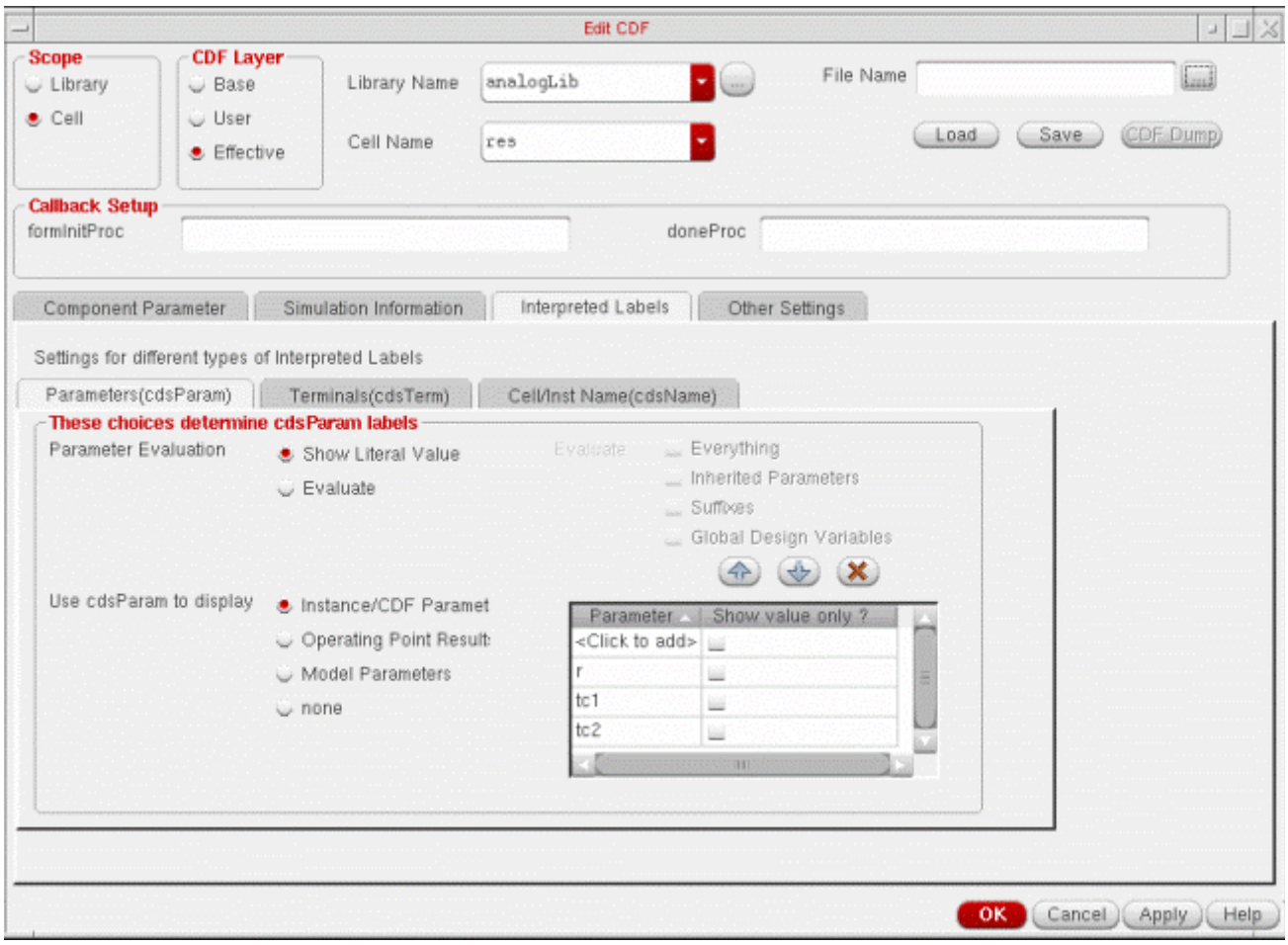
You can set the default label display options in the Interpreted Labels tab of the Edit CDF form. The Interpreted Labels tab has the following three tabs that are described below:

- Parameters(cdsParam)
- Terminals(cdsTerm)
- Cell/Inst Name(cdsName)

Library label information differs from cell label information. So the display that you see in Parameters(cdsParam), Terminals(cdsTerm) and Cell/Inst Name(cdsName) tabs depends on whether you are viewing the CDF for a library or a cell. For more information about the tabs, see the following sections.

Parameters(cdsParam)

The *cdsParam*( ) labels (usually placed on the layer *annotate drawing*) display information about parameter values or backannotated parameter values. You can set the default display options for cdsParam labels in the Parameters(cdsParam) tab.



The fields in the Parameters(cdsParam) tab are described below:

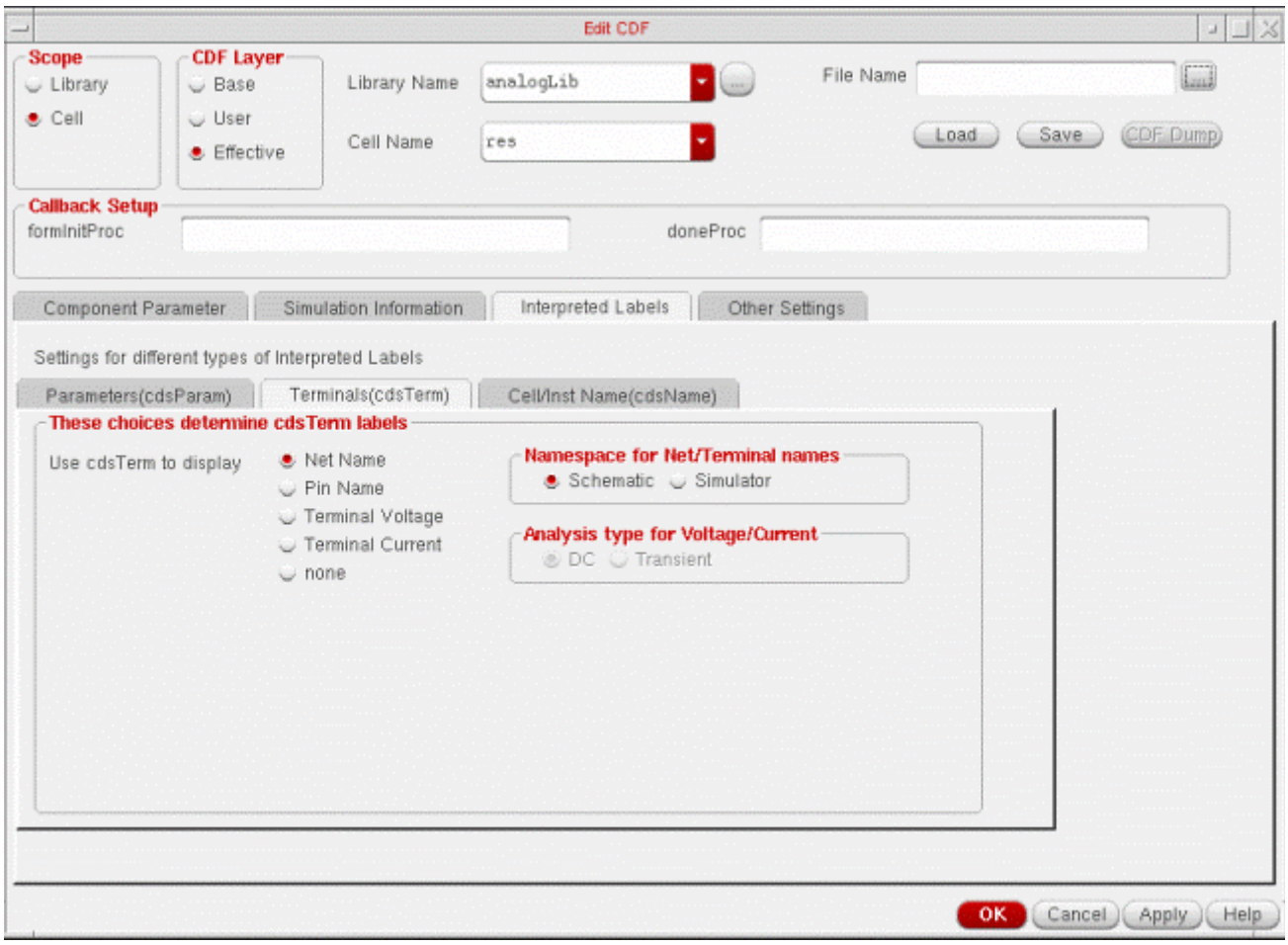
- **Parameter Evaluation** selects a method for displaying parameter values. (These different methods of display are necessary because a parameter value can be a constant, a design variable, an expression using constants or design variables, or a value or expression inherited from the hierarchy.)
  - **Show Literal Value** displays the literal parameter value as entered.
  - **Evaluate** evaluates the parameter value as specified in the *Evaluate* field.

- **Evaluate** selects the parameter evaluation method if *Parameter Evaluation* is set to *Evaluate*. Select the check box next to one of the following options:
  - **Everything** performs complete numeric evaluation of the expression and displays the result. For example, if the value is *Rin*10* and *Rin* is 5, it displays *50*. If the value is *pPar(Rin)*100* and the parent instance has a parameter *Rin* set to 10, it displays *1000*.
  - **Inherited Parameters** evaluates inherited parameters in expressions. For example, if the value is *pPar(Rin)*100* and the parent instance has a parameter *Rin* set to 10, the label displays *10 * 100*.
  - **Suffixes** converts suffixes in the parameter value to numerical values. For example, 1K becomes 1000.
  - **Global Design Variables** evaluates global design variables in expressions. For example, if the value is *Rin*100* and *Rin* is 5, the label displays *5 * 100*.
- **Use cdsParam to display** allows you to select the kind of parameters to display for cells.
  - **Instance/CDF Parameter** displays the instance or CDF parameters listed in the table. For information about selecting the parameters to be displayed, see Specifying cdsParam Parameters to Display on page 110.
  - **Operating Point Result** displays the DC or transient operating point results from simulation for the parameters listed in the table. For information about selecting the parameters to be displayed, see Specifying cdsParam Parameters to Display on page 110.
  - **Model Parameters** displays the simulation values for the model parameters listed in the table. For information about selecting the parameters to be displayed, see Specifying cdsParam Parameters to Display on page 110.
  - **none** displays no parameters.

  Note: The **Use cdsParam to display** fields are not applicable to library-level CDF.
- **Operating Point Results Type** selects the simulation type for the operating point result that you want to annotate. The choices are *DC* or *Transient*. This field is for operating point parameters only.

Terminals(cdsTerm)

The *cdsTerm( )* labels (usually placed on the layer *annotate drawing8*) display information about the pin or about the nets attached to the pin. You can set the default display options for cdsTerm labels in the Terminals(cdsTerm) tab.



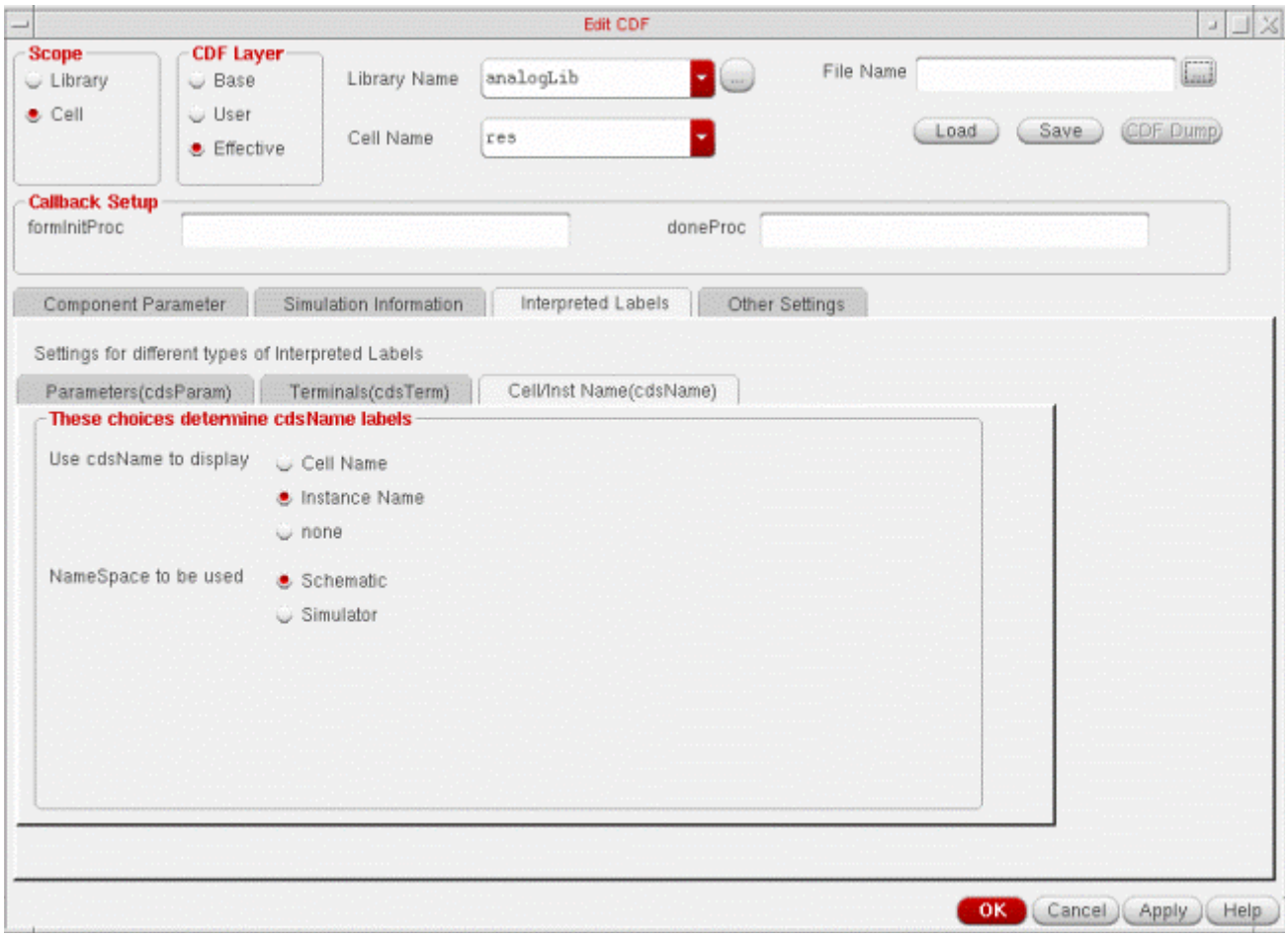
The fields in the Terminals(cdsTerm) tab are described below:

- **Use cdsTerm to display** selects what is displayed next to the component terminals.
  - **Net Name** displays the terminal net names.
  - **Pin Name** displays the terminal pin names.
  - **Terminal Voltage** displays the terminal voltage.
  - **Terminal Current** displays the terminal current.
  - **none** does not display cdsTerm labels.

- **Namespace for Net/Terminal names** selects the type of net name to display if *Use cdsTerm to display* is set to *Net Name* or *Pin Name*. The choices are *Schematic or Simulator*.
- **Analysis type for Voltage/Current** selects the simulation type for the result that you want to annotate. The choices are *DC or Transient*. This field is used only when *Use cdsTerm to display* is either *Terminal Voltage* or *Terminal Current*.

Cell/Inst Name(cdsName)

The *cdsName( )* labels (usually placed on the layer *annotate drawing7*) display information about the cell name or the instance name. You can set the default display options for cdsName labels in the Cell/Inst Name(cdsName) tab.



The fields in the Cell/Inst Name(cdsName) tab are described below:

- **Use cdsName to display** selects what is displayed next to the component symbol.

- **Cell Name** displays the cell name of the component.
- **Instance Name** displays the instance name of the component.
- **none** specifies that no information is displayed.

- **NameSpace to be used** selects the type of instance name. The choices are *Schematic or Simulator*. This field is used only when the *Use cdsName to display* is *Instance Name*.

Creating Labels Using the Edit CDF Form

You can make selections for label information for both library and cell CDFs. You can use the following procedure to create labels for a library CDF.

1. Click the Interpreted Labels tab of the Edit CDF form.
2. Make edits for library or cell CDF label specifications.
3. Click *Apply* to apply these specifications to the current CDF.

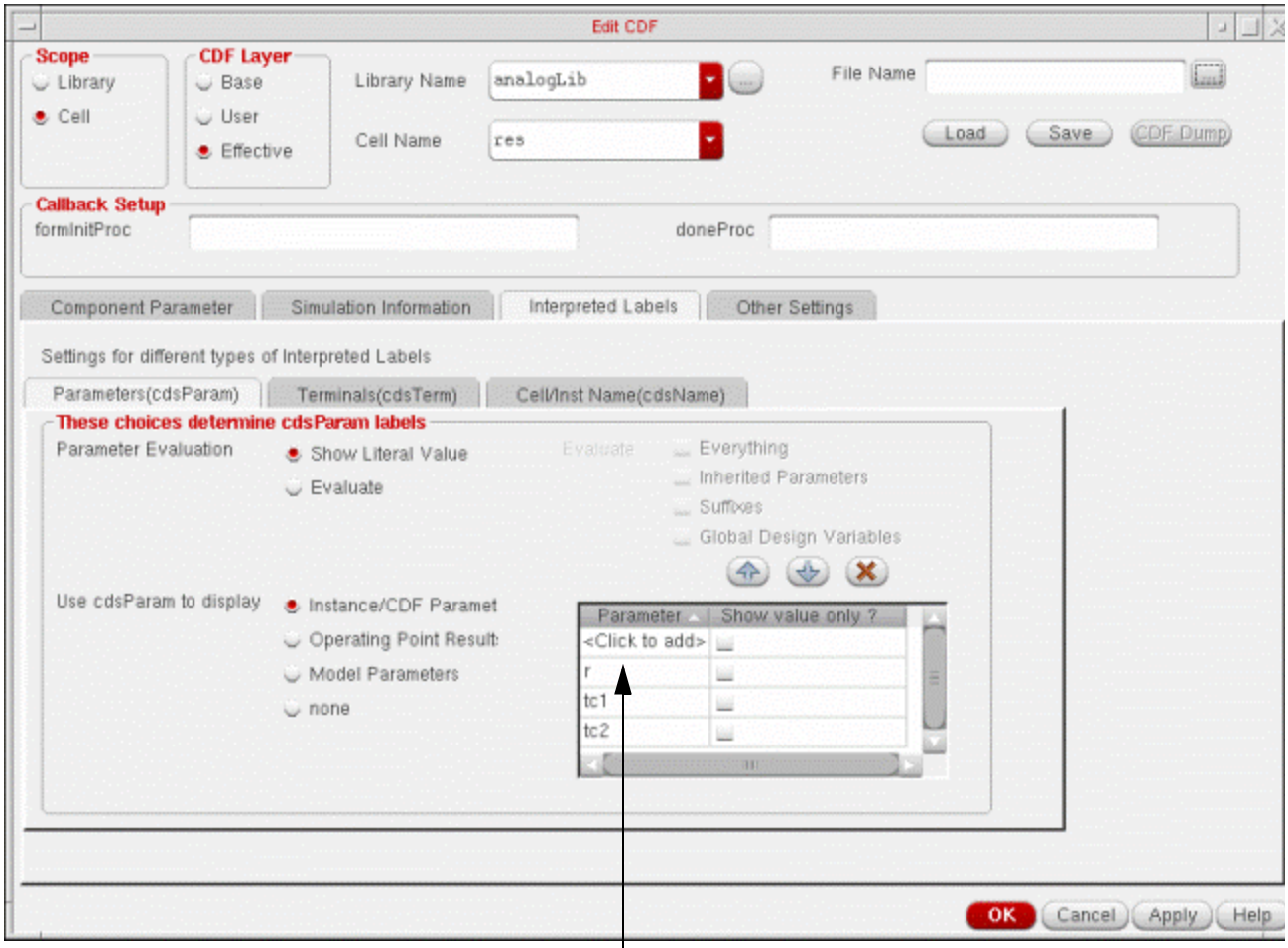
Specifying cdsParam Parameters to Display

The Parameters(cdsParam) tab on the Interpreted Labels tab of the Edit CDF form allows you to specify the parameters to be displayed for cdsParam labels.

To specify the cdsParam parameters to display, do the following:

1. Select the radio button corresponding to the parameters you want to display—*Instance/CDF Parameter*, *Operating Point Result* or *Model Parameters*.

The table on the right lists the parameters to be displayed. For example, the table in the following figure lists the parameters to be displayed if the *Instance/CDF Parameter* radio button is selected.



Parameters to be displayed for cdsParam labels

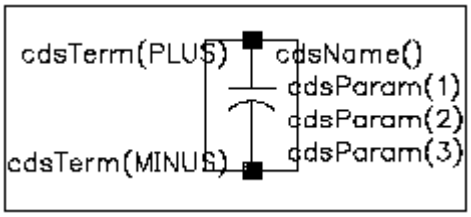
2. Specify the parameters you want to display:

- To add a parameter, click where it says *<Click to add>* in the *Parameter* column and type the name of the parameter.

  Note: The new parameter is also displayed in the Component Parameter tab.

- The parameters are displayed in the order in which they are listed in the table. The number of parameters that are displayed is also limited to the number of cdsParam

labels on the cell. For example, the following figure shows the symbol for a capacitor that has three cdsParam labels. For this cell, only the first three parameters listed in the table are displayed.



To change the order in which the parameters are listed in the table, select a parameter and click the  or  buttons.

- By default, a parameter name and its value are displayed in the format:
  `parameter name : value`

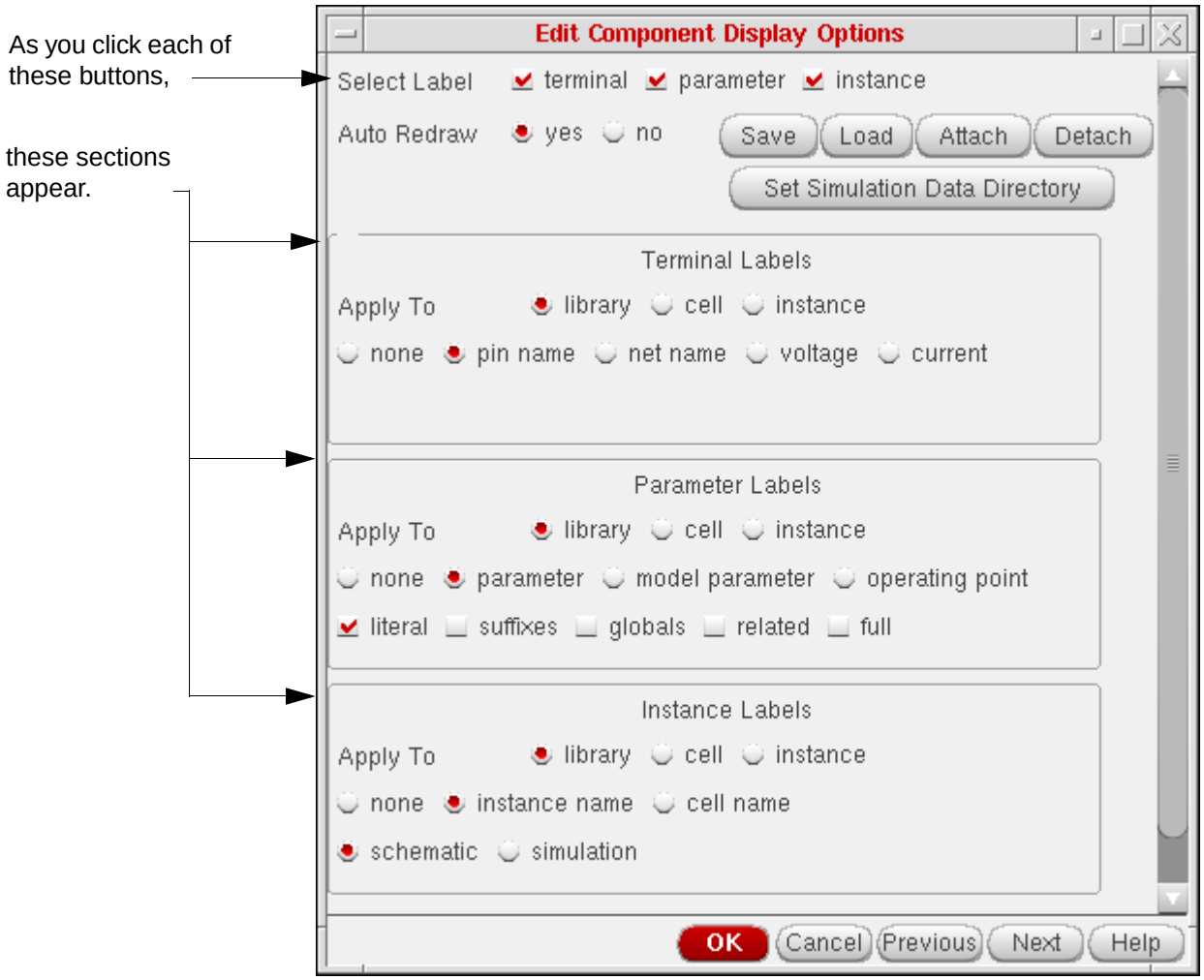
  To display only the value of a parameter, select the check box next to the parameter in the *Show Value Only ?* column.

- To delete a parameter, select the parameter and click the  button.

3. Click *Apply*.

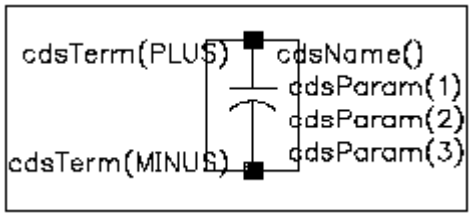
The Edit Component Display Options Form

You can control the creation of CDF labels by using the *Component Display* command in the *Edit* menu in the schematic editor to open the Edit Component Display Options form. Each section shown below is displayed when you click on the corresponding *Select Label* button.



When you change the display of information in the Edit Component Display Options form, the display of *ilLabels* (labels generated automatically during automatic symbol generation) and interpreted labels is affected. *ilLabels* are usually defined on symbols and are displayed

when you place the symbol in a schematic. For example, the following figure shows the symbol for a capacitor from *analogLib*:



The *cdsTerm( )* labels (usually placed on the layer *annotate drawing8*) display information about the pin or about the nets attached to the pin. These labels must be of the form *cdsTerm(pinName)*. In the Edit Component Display Options form, you use the Terminal Labels settings to control the *cdsTerm* labels.

If *pinName* contains special characters, you must quote or escape them properly.

The *cdsName( )* labels (usually placed on the layer *annotate drawing7*) display information about the cell name or the instance name. In the Edit Component Display Options form, you use the Instance Labels settings to control the *cdsName* label.

The *cdsParam*( ) labels (usually placed on the layer *annotate drawing*) display information about parameter values or backannotated parameter values. Three labels are usually generated during automatic symbol generation, but you can define additional labels. The only requirement for the parameter labels is that they are sequential, starting with 1 if they are numbers. In the Edit Component Display Options form, you use the Parameter Labels settings to control the *cdsParam* labels.

Note: *cdsParam*( ) can also take parameter names. Therefore, as *cdsParam(`n`)*, you can use *cdsParam(`s_paramName`),* for example, cdsParam('pm), where pm is a parameter name. You must declare the parameter names in the *Instance/CDF Parameter* field in the Parameters(cdsParam) tab on the Interpreted Labels tab of the Edit CDF form. The cdsParam points to those parameters in an orderly manner. If you change the order of those parameters, the displayed values will be changed.

The paramLabelSet property determines which properties of the cell you would like to see on the schematic. When paramLabelSet = nil (default), then cdsParam ("paramname") retrieves its value. When you set paramLabelSet explicitly to a blank string ("""), no parameter will be retrieved and displayed on the schematic by cdsParam. This is because you are forcefully asking the system not to retrieve any parameter in this case.

In the Edit Component Display Options form, there are three cyclic fields for parameter name labels. If you add a fourth parameter label to your cell, a fourth cyclic field appears on the Edit Component Display Options form. The syntax for the fourth parameter label is *cdsParam(4)*.

If you choose the *library* option in the *Apply To* field in the Edit Component Display Options form, the display information is written to the library user-level CDF. If you choose the *cell* option, the display information is written to the cell user-level CDF. If you choose the *instance* option, the display information is saved as properties on that instance. Only the instance label display is saved when you close the library because user-level CDF data applies only while the library is open. (It is not written to the database or saved when you exit the Cadence® software.) To keep this library and cell CDF information, copy it into a library or base CDF description that is permanently saved.

Creating Labels in the Schematic Editor

You can set up label display with the Edit Component Display Options form and the Edit CDF form as follows:

1. Set up your component and place it in a schematic.
2. Choose *Edit–>Component Display* to set up the labels you want placed with the component.

   The Edit Component Display Options form appears.
3. On the Edit CDF form, look at the effective-level CDF description for the labeled component.

   The effective-level cell CDF is the combination of the base-level CDF (with all parameter and simulation information defined) and the user-level CDF that you created with the Edit Component Display form.
4. Type in a filename and click *Save* to save the contents of the form in a file.
5. Change the CDF type to *base*.
6. Enter the same filename and click *Load*.

   The effective-level CDF is copied to a base-level CDF with all of the interpreted label information set up as you saw it.
7. Click *OK* to save this new base CDF description.
8. Repeat this procedure to create a library CDF description.

When you place a component in the schematic editor, the Add Instance form shows the parameters associated with this component. No CDF properties (labels) appear unless you assign parameter values other than the default values. If you want to see the value of any parameter displayed in the design window, you can toggle the display setting for that parameter to make it visible.

Creating Labels with SKILL

You can use the following Cadence SKILL language functions with the right arrow operator (`->`) to select label specifications. You enter the specifications as properties of either the library or cell CDF. Not all properties need to be set. Enter only those properties that you want set.

The following SKILL expressions set label CDF specifications at the library level:

- paramDisplayMode (equivalent to *Use cdsParam to display* setting in the Parameters(cdsParam) tab)

  ```
  libId = ddGetObj("analogLib")
  cdfId = cdfCreateBaseLibCDF(libId)
  cdfId->paramDisplayMode = “parameter”
  ```

- paramEvaluate (equivalent to *Evaluate* setting in the Parameters(cdsParam) tab)

  ```
  cdfId->paramEvaluate = "t nil nil nil nil"
  ```

  For the paramEvaluate option list, the first entry applies to literal, the second to suffixes, the third to globals, the fourth to inheritance, and the fifth to full.

- paramSimType (equivalent to *Operating Point Results Type* setting in the Parameters(cdsParam) tab)

  ```
  cdfId->paramSimType = "DC"
  ```

- termDisplayMode (equivalent to *Use cdsTerm to display* setting in the Terminals(cdsTerm) tab)

  ```
  cdfId->termDisplayMode = "netName"
  ```

- termSimType (equivalent to *Analysis type for Voltage/Current* setting in the Terminals(cdsTerm) tab)

  ```
  cdfId->termSimType = "DC"
  ```

- netNameType (equivalent to *Namespace for Net/Terminal names* setting in the Terminals(cdsTerm) tab)

  ```
  cdfId->netNameType = "schematic"
  ```

- instDisplayMode (equivalent to *Use cdsName to display* setting in the Cell/Inst Name(cdsName) tab)

  ```
  cdfId->instDisplayMode = "instName"
  ```

- `instNameType (equivalent to` *Namespace to be used* setting in the Cell/Inst Name(cdsName) tab)

  ```
  cdfId->instNameType = "schematic"
  ```

The following expressions set label CDF specifications at the cell level:

- paramLabelSet (equivalent to *Instance/CDF Parameters* setting in the Parameters(cdsParam) tab)

```
cell = ddGetObj(libId "NFET")
cdfId = cdfCreateBaseCellCDF(cell)
cdfId->paramLabelSet = "-model l w"
```

- opPointLabelSet equivalent to *Operating Point Results* setting in the Parameters(cdsParam) tab)

```
cdfId->opPointLabelSet = "id vgs vds"
```

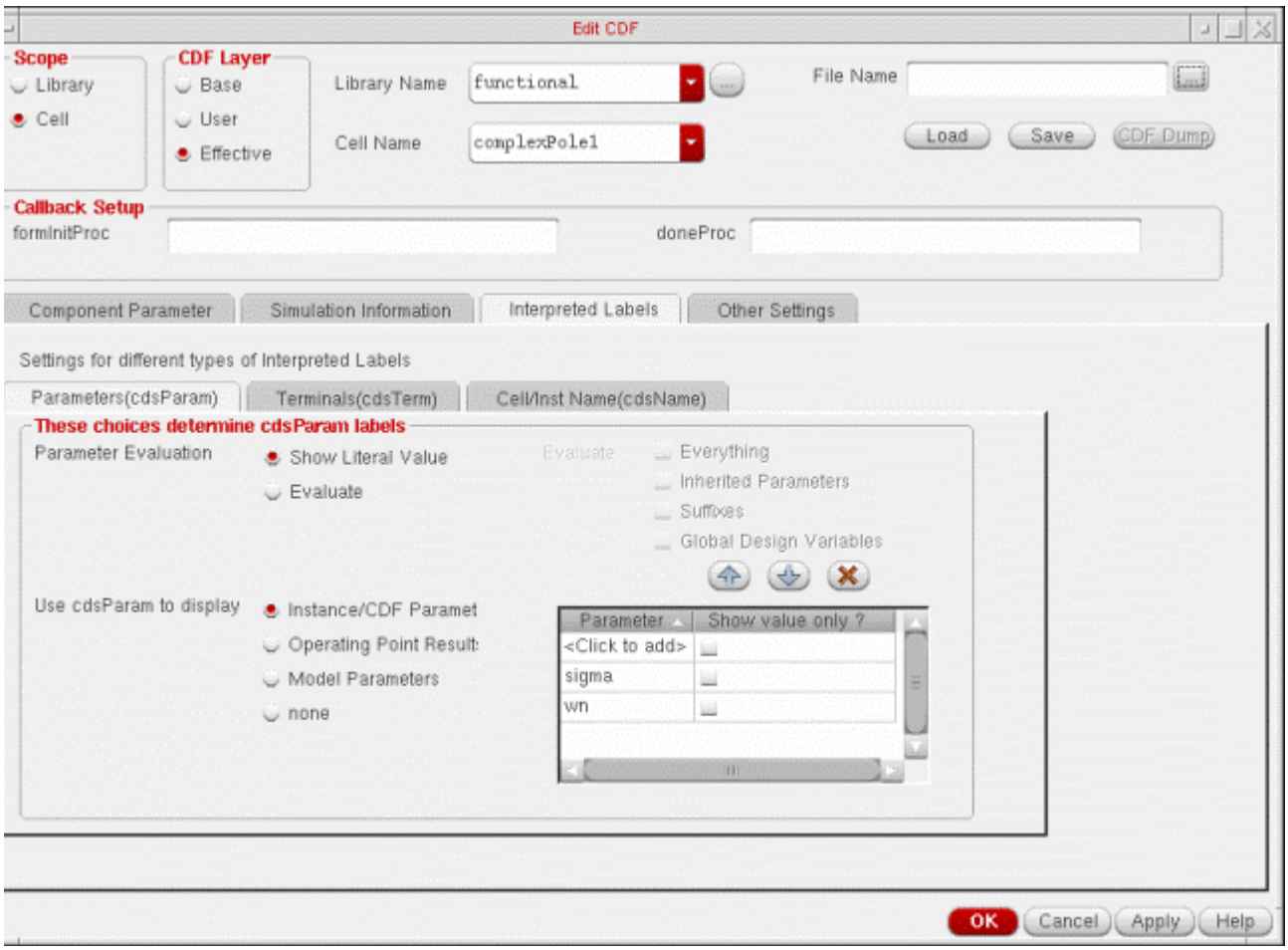
- modelLabelSet (equivalent to *Model Parameters* setting in the Parameters(cdsParam) tab)

```
cdfId->modelLabelSet = "vfb phi eta"
```

Complex Pole Example

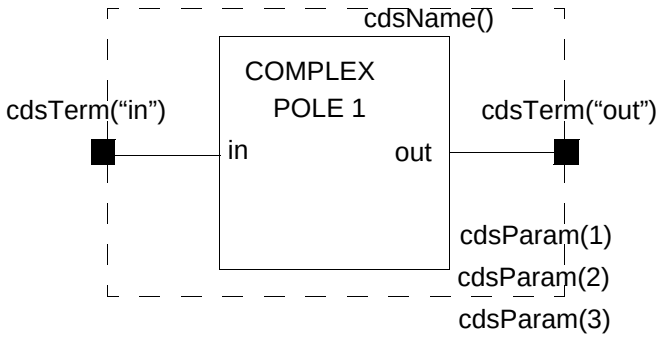
In previous chapters, you used the Edit CDF form on the *complexPole1* component to look at the Component Parameter and Simulation Information sections of the CDF description. Now you look at the Interpreted Labels information in the complex pole CDF description and see how it affects the display of the component's symbol.

The following figure shows the Interpreted Labels tab of the Edit CDF form for *complexPole1*.



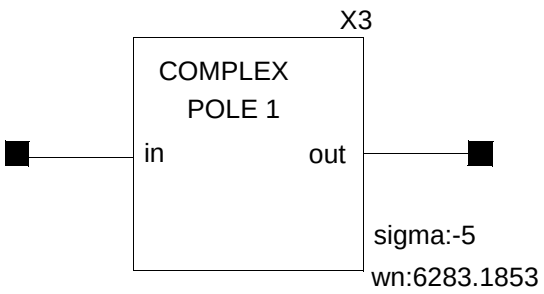
1. Look at the *complexPole1* symbol in the Symbol Editor. (Open the complexPole1 design, specifying the symbol cellview.)

   You see a generic version of the component, with no values displayed.



2. Open an empty schematic editor window by creating a new design.
3. Use the *Add Instance* command to add a *complexPole1* component to the schematic.

   Its symbol looks like this:



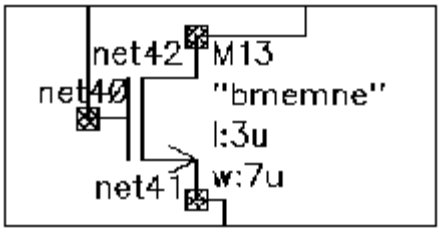
The two default parameter values from the Component Parameters tab and the instance name are set to be displayed (with the *Use cdsParam to display* setting in the Parameters(cdsParam) tab and the *Use cdsName to display* setting in the Cell/Inst Name(cdsName) tab). However, the *Use cdsTerm to display* setting in the Terminals(cdsTerm) tab is set to *Net Name*, and the component is by itself, not connected to any nets. There are no net names to display on the terminals (*in* and *out*).

NFET Example

The NFET symbol in the following figure shows the effect of setting *Use cdsParam to display* setting in the Parameters(cdsParam) tab to *Instance/CDF Parameters* with the *model*, *l* and *w* parameters. Three parameters, *model*, *l*, and *w,* are displayed. There is no parameter name displayed with *bmemne* (the model name) because the *Show Value Only* check box next to the *model* parameter is selected in the Parameters(cdsParam) tab.

The default values from the CDF description are displayed for the *l* and *w* parameters, as shown by the format of *parameter name : value* (*l:3u* and *w:7u*). In addition, the *Use cdsTerm to display* setting in the Terminals(cdsTerm) tab is set to *Net Name* and the *Use cdsName to display* setting in the Cell/Inst Name(cdsName) tab is set to *Instance Name*.

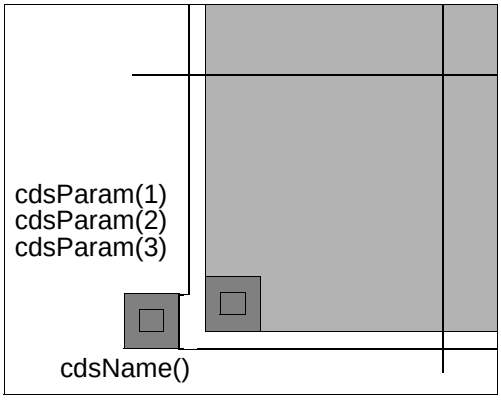
Unlike the complex pole example, this transistor is connected to nets in a schematic, so the net and instance names are all displayed.



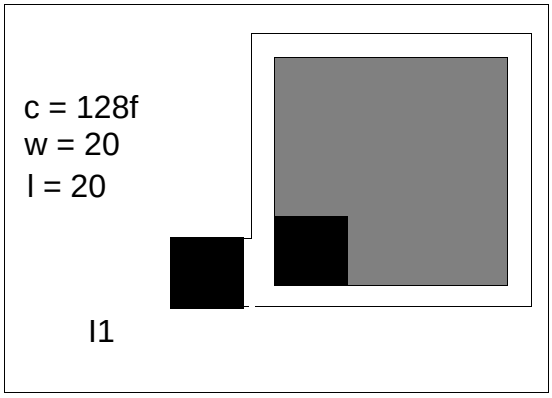
Try changing the fields to different settings and then view the results.

You can use labels in any view. For example, you can set up a layout with labels so that you can see instance and parameter information when you place an instance:

Layout Master



Instance



This can be especially useful when you build the *ivpcell* used in layout extraction. In this way, when you look at the extracted layout, you see the instance found and the parameters measured.

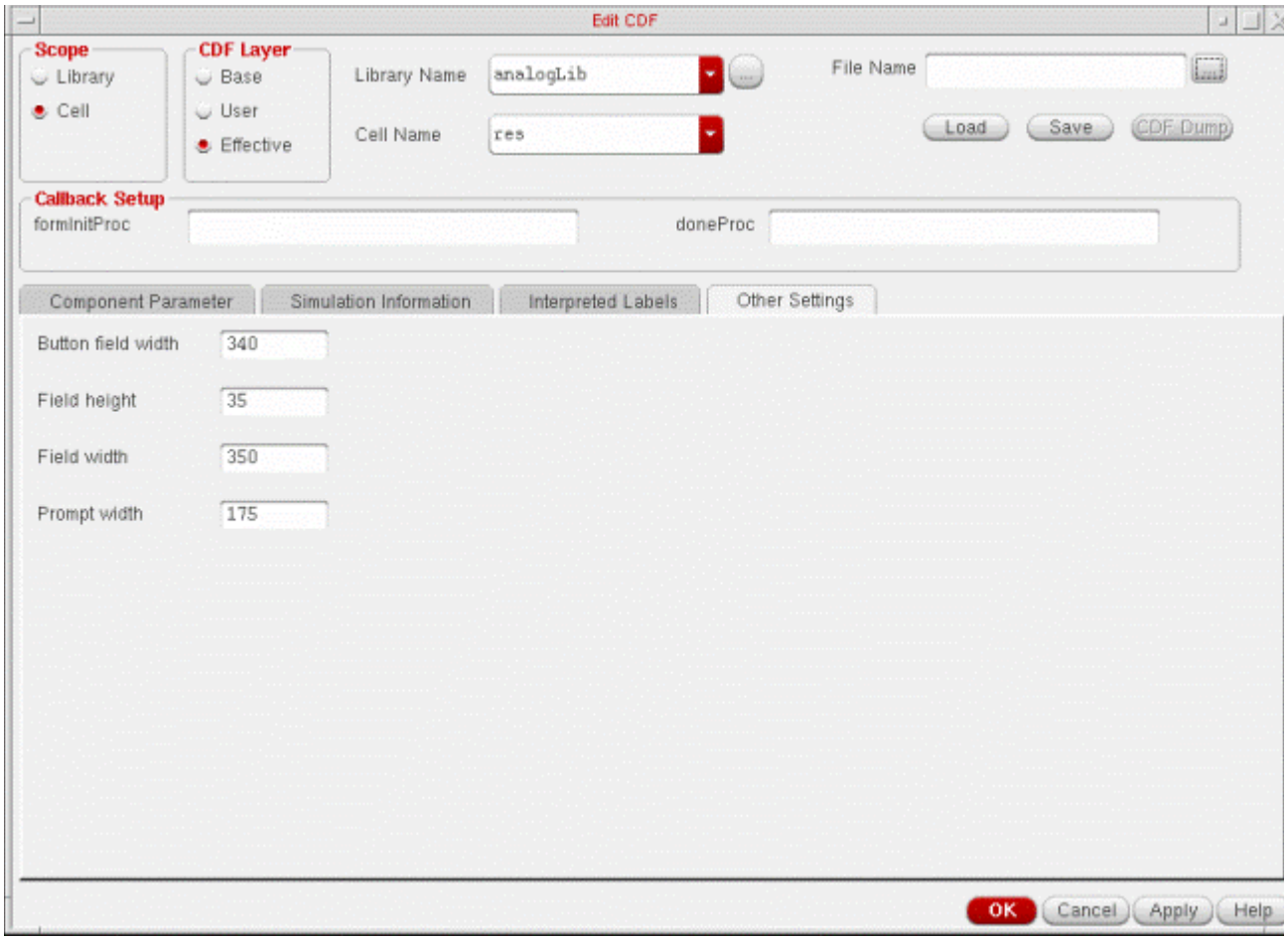
Note: These labels do not pick up hierarchical information, so *cdsName* will only show the current name, not the hierarchical name.

6

Other CDF Information

Other Settings Tab

You use the Other Settings tab of the Edit CDF form to specify the sizes of input fields.



The following fields let you control the size of parameters on an instantiation form (such as the Add Instance or Edit Object Properties forms). These fields do not affect the display in the Edit CDF form.

Button field width lets you specify the width of a button on the instantiation form. The default width is 340 pixels.

Field height lets you specify the height of a field on the instantiation form. The default height is 35 pixels.

Field width lets you specify the width of a field on the instantiation form. The default width is 350 pixels.

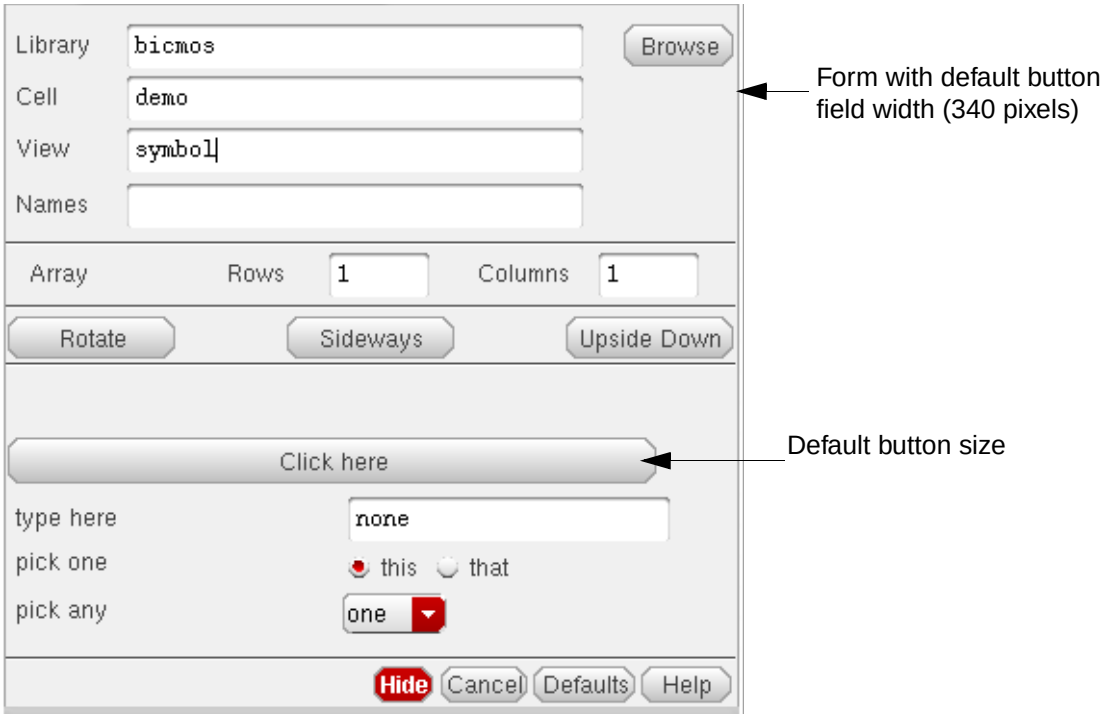
Prompt width lets you specify the width of the prompt on the instantiation form. The default width is 175 pixels.

Setting Up Fields and Prompts

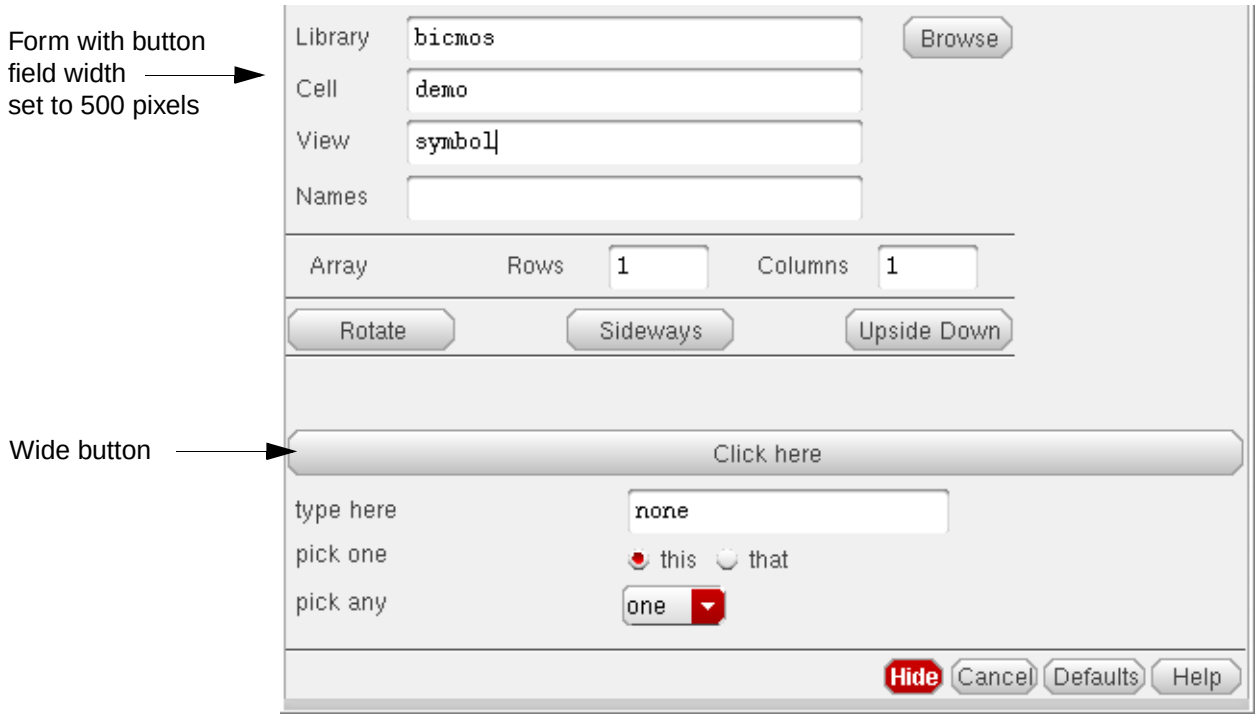
The input fields and prompts for a component are displayed on an instantiation form, such as the Add Instance or Edit Object Properties forms. The following examples show the Add Instance form as it appears when a component is instantiated. The sample CDF description in these examples defines four parameters: a button, a string, a cyclic field, and a radio button field.

Adjusting Button Fields

You can adjust the width of a field with the *Button field width* parameter in the Other Settings tab on the Edit CDF form. The following examples show button fields on two Add Instance forms. The following form shows the button with the default width of 340 pixels.

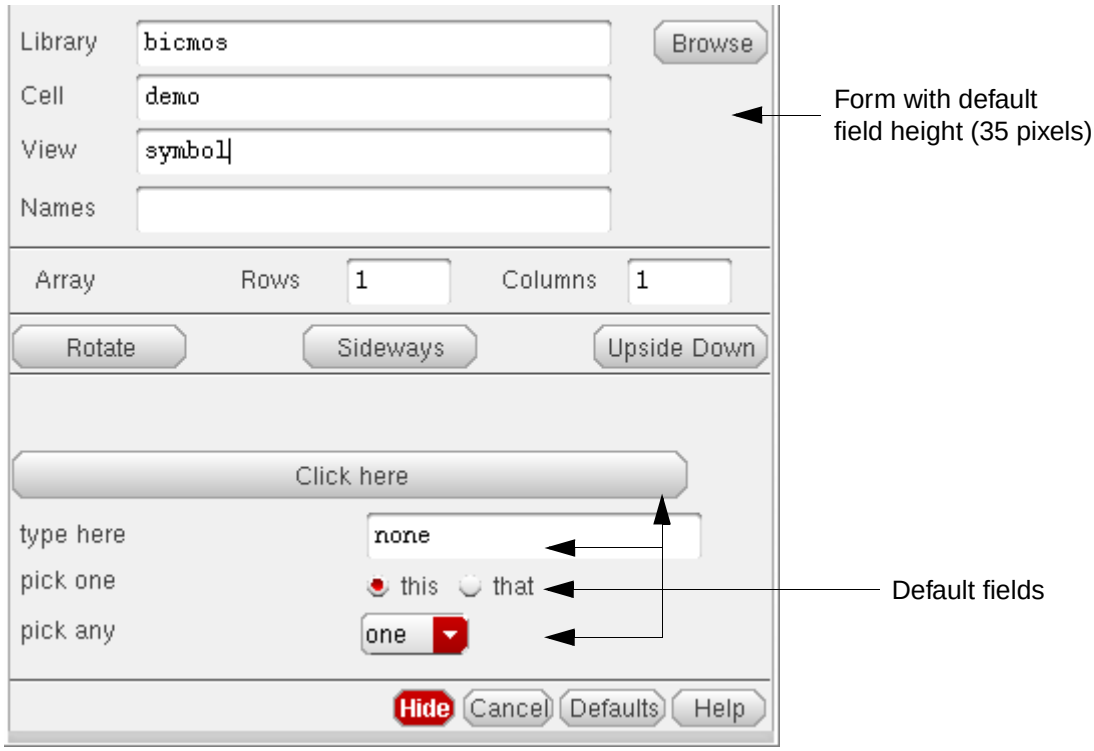


The following form shows the button with a width of 500 pixels.

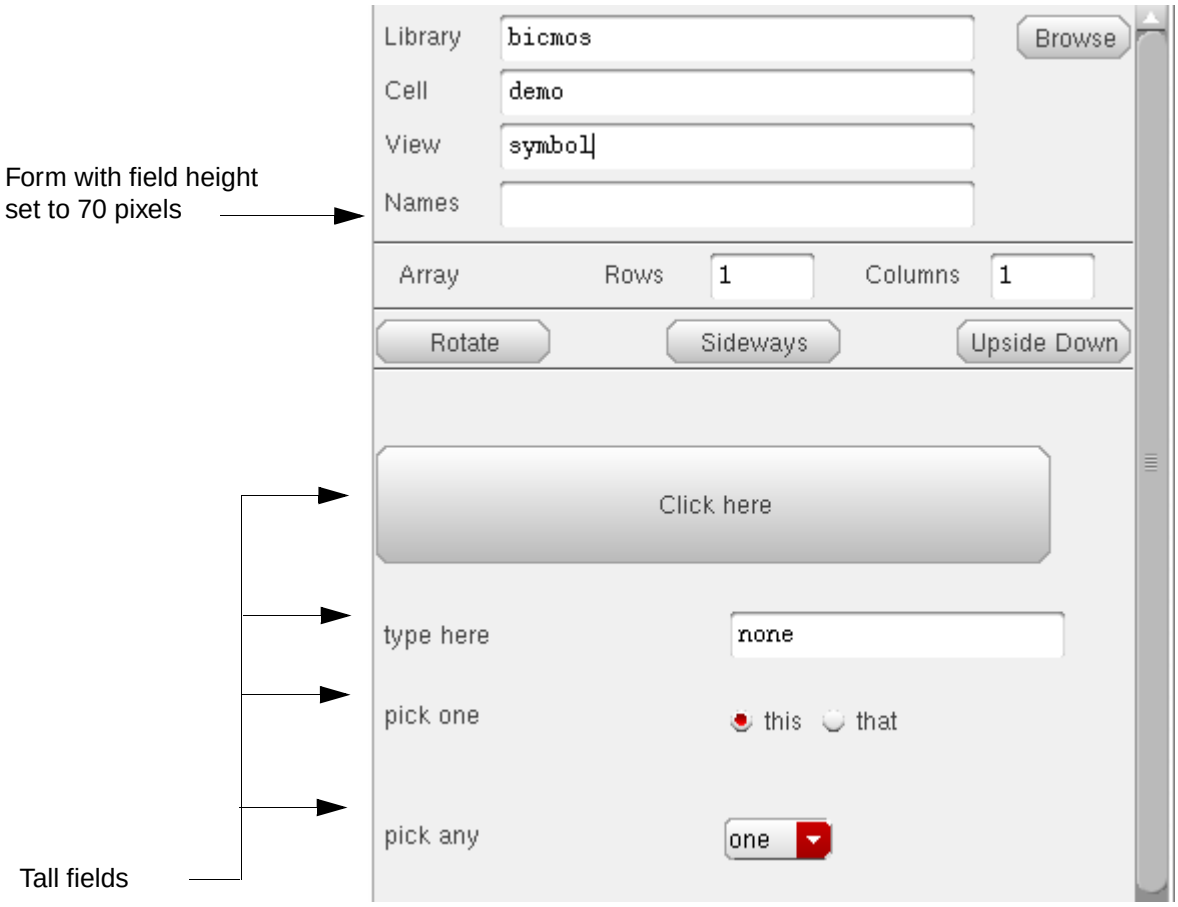


Adjusting Field Heights

You can adjust the height of a field with the *Field height* parameter in the Other Settings tab on the Edit CDF form. The following example shows fields on two Add Instance forms. The following form shows the fields at the default height of 35 pixels.



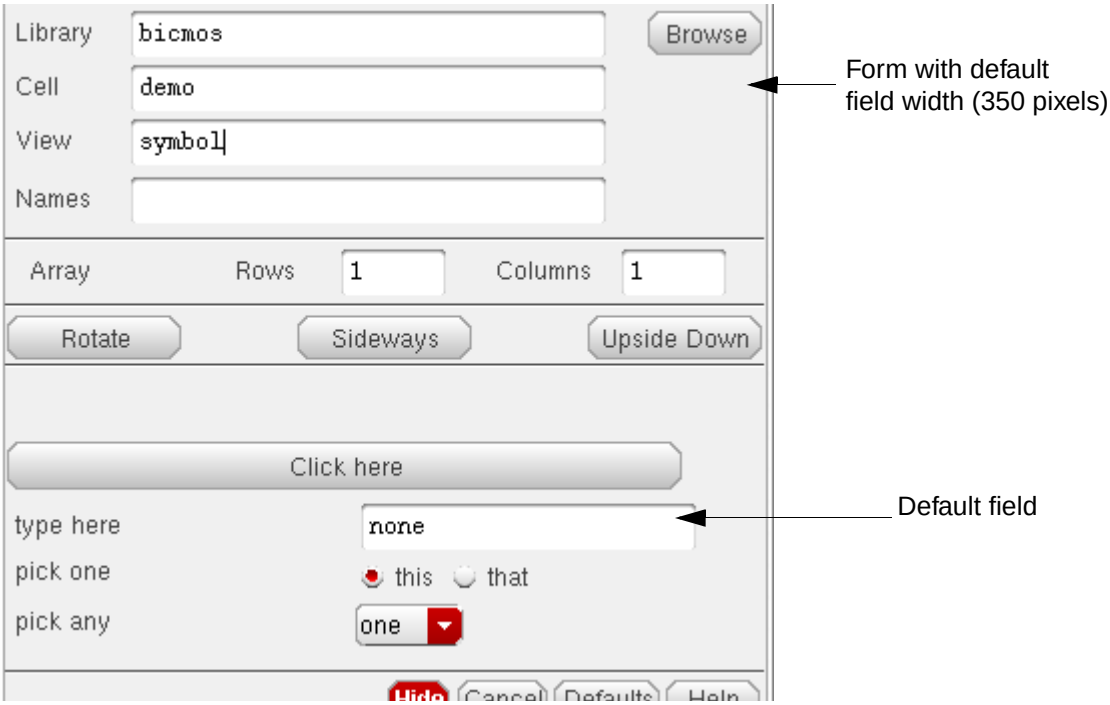
The following form shows the field height at 70 pixels.



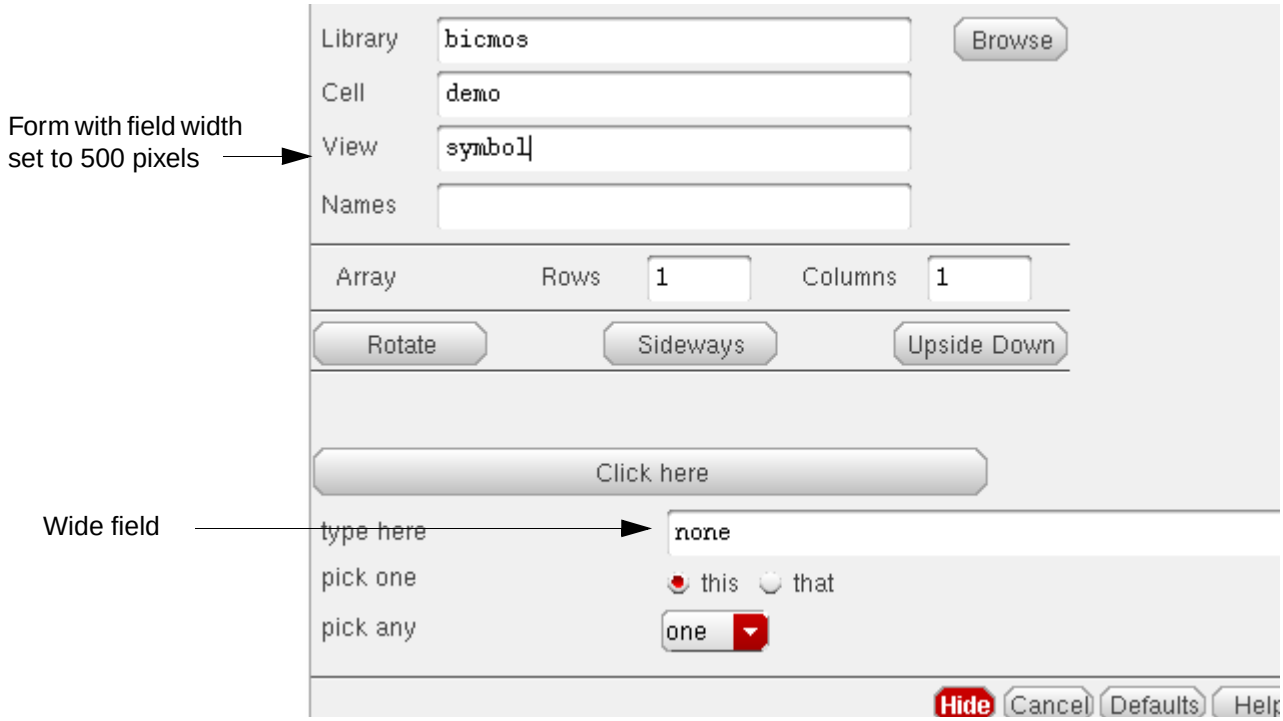
Adjusting Field Widths

You can adjust the width of a field with the *Field width* parameter in the Other Settings tab on the Edit CDF form. This parameter adjusts the width of string input fields only. (You adjust the width of a button field with the *Button field width* parameter in the Other Settings tab on

the Edit CDF form.) The following examples show string input fields on two Add Instance forms. The following form shows the field at the default width of 350 pixels.

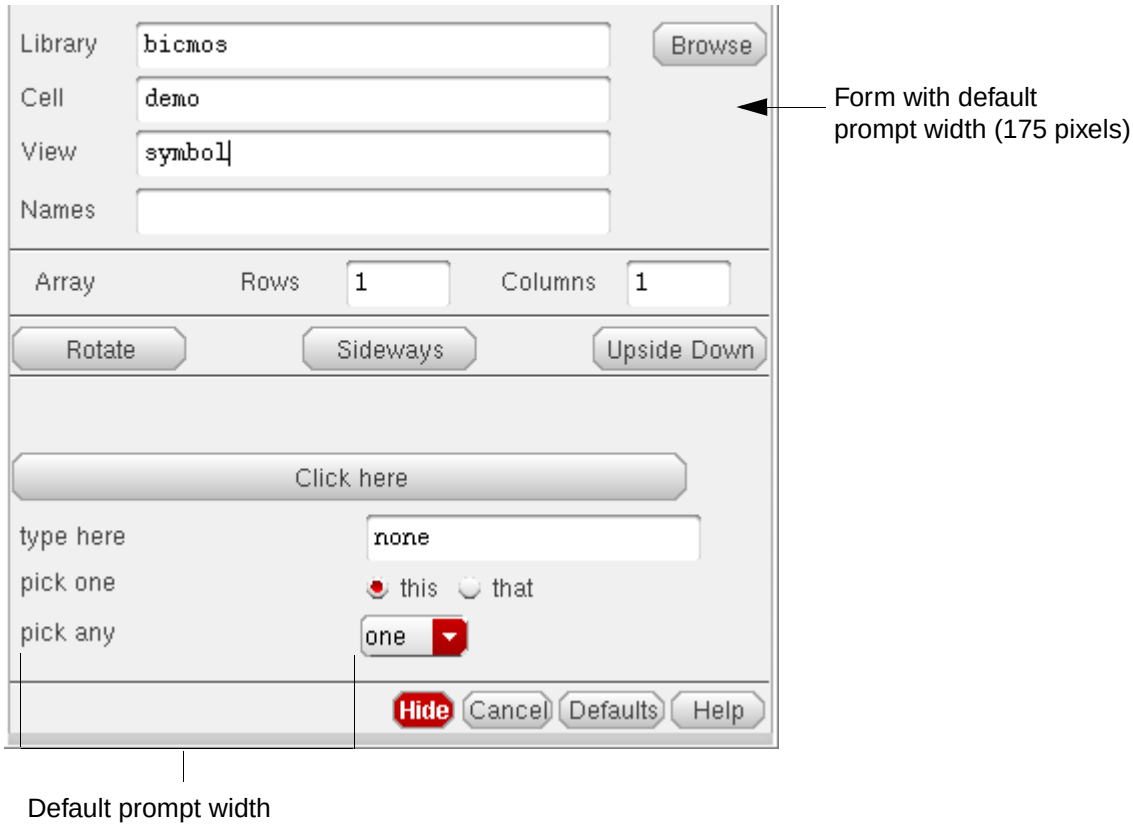


The following form shows a field width of 500 pixels.

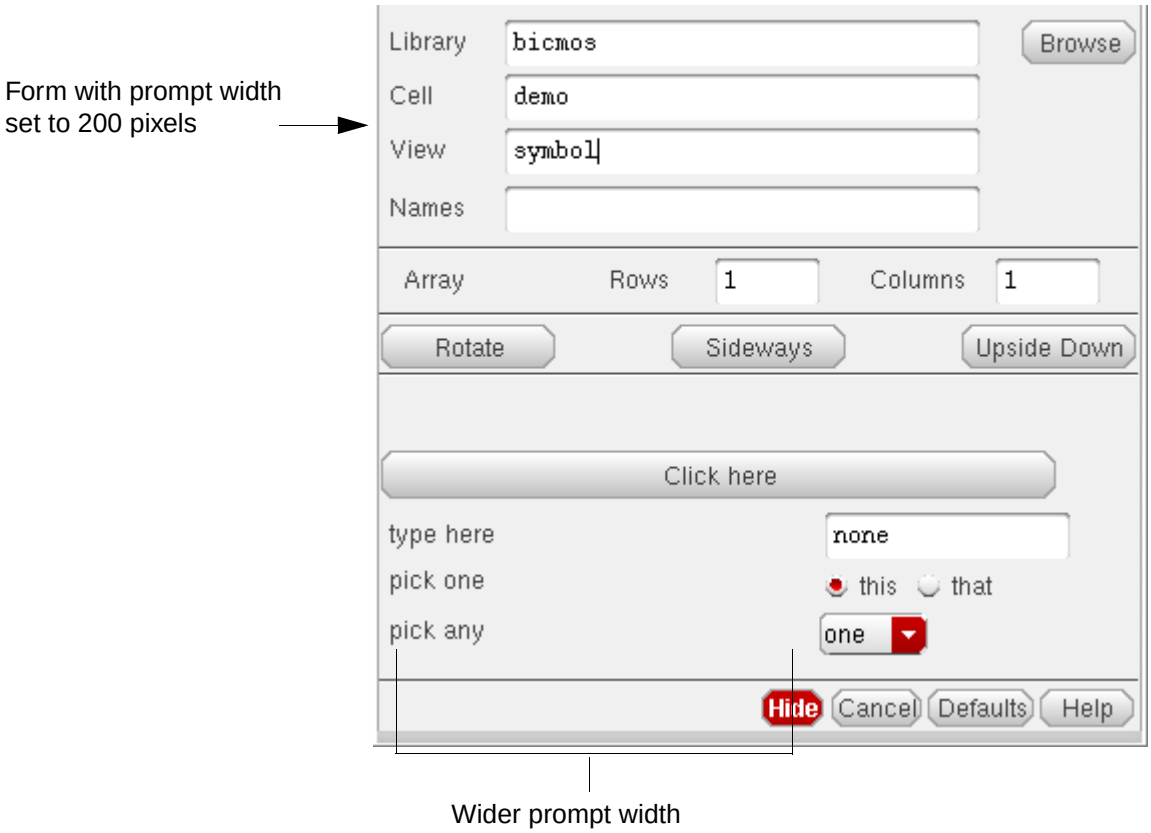


Adjusting Prompt Widths

You can adjust the width of a prompt with the *Prompt width* parameter in the Other Settings tab on the Edit CDF form. The following example shows prompts on two Add Instance forms. The following form shows the prompts at the default width of 175 pixels.



The following form shows the prompt width of 200 pixels.



Initialization Procedure

The *formInitProc* field in the header of the Edit CDF form lets you enter the name of an optional procedure for preprocessing CDF data. The procedure executes when the contents of the CDF are displayed on a form, such as the Add Instance form or the Edit Object Properties form. The procedure takes a single argument, *cdfDataId*, which is the database object that represents the CDF description for the component being loaded into the form. You enter only the function name (with no parentheses) in the *formInitProc* field.

In the NFET example, you might enter an initialization procedure to reset all NFET parameters to their defaults each time the component is displayed on a form. (This is only an example. It is not practical to reset parameters to their default values every time you return to the Edit Object Properties form.)

Enter the procedure name, *setDefaults*, in the *formInitProc* field. This procedure takes only one argument, *cdfDataId*, which is used to point to the parameters and values of the associated component.

You can create the SKILL procedure with a standard editor. You must load the file in order for the initialization procedure to be called and run. (See "Loading Callbacks" on page 139 for details about loading or attaching a SKILL procedure.)

```
procedure(setDefaults(cdfDataId)
        ;used as formInitProc to set all nfet values to the
        ;default values as form is entered
    cdfDataId->w->value = cdfDataId->w->defValue
    cdfDataId->width->value = cdfDataId->width->defValue
    cdfDataId->l->value = cdfDataId->l->defValue
    cdfDataId->length->value=cdfDataId->length->defValue
)
```

An initialization procedure runs every time the form is updated. The procedure runs when you change a value on the component (a callback), and when you point to a new component.

You can also enter the following SKILL expression to set the *formInitProc* property when you create the cell CDF:

```
cdfId = cdfCreateBaseCellCDF( cellId )
.
.
.
cdfId->formInitProc = "setDefaults"
```

Postprocessing Procedure

The *doneProc* field in the header of the Edit CDF form lets you enter the name of an optional procedure to be executed after you change a parameter on a component instance. You execute the procedure when you click on *Apply* on the Edit Object Properties form or when you place an instance of the component using the Add Instance form.

For the NFET example, a *DoneProc* procedure might recalculate the simulation parameters every time the instance changes. Enter the procedure name in the *DoneProc* field. In the following example, the procedure name is *calcSimParams*. This procedure takes a single argument, *cellId*, that identifies the cell of the instance that was modified. Throughout the procedure, *cdfgData*, the global variable for CDF information, accesses the CDF information.

You can create the SKILL procedure with a standard editor. You must load this file in order for the procedure to be called and run. (See "Loading Callbacks" on page 139 for details about loading or attaching a SKILL procedure.)

```
procedure(calcSimParams(cellId)
    cdfgData = cdfGetInstCDF(cellId)
        tmp=cdfParseFloatString(cdfgData->w->value)
        cdfgData->ad->value=sprintf(s "%g" (tmp * 7.0 * 1e-6))
```

```
        cdfgData->as->value=sprintf(s "%g" (tmp * 7.0 * 1e-6))
        cdfgData->pd->value=sprintf(s "%g" ((tmp + 7e-6) * 2))
        cdfgData->ps->value=sprintf(s "%g" ((tmp + 7e-6) * 2))
)
```

You can also enter the following SKILL expression to set a postprocessing procedure when you create the cell CDF:

```
cdfId = cdfCreateBaseCellCDF( cellId )
...
cdfId->doneProc = “calcSimParams”
```

7

Writing Callbacks

Overview

A callback procedure is a Cadence® SKILL language expression that is evaluated when a parameter value changes. Using callbacks is a powerful way to control relationships between parameters and the restrictions on the parameter itself.

Note: Cadence advises you to have sufficient knowledge of callback procedures before using them.

Callbacks can be GUI based or database based. GUI based callbacks occur when you modify the values in the parameter form fields. Database based callbacks occur when the parameter value is modified via a database action. The CDF parameter callback is primarily a GUI based callback. A GUI based callback is active when the CDF parameters are displayed in the Add Instance form or the Edit Object Properties form when you use the Create Instance or Edit Properties commands.

Note: The CDF parameter callback is invoked only when you use the Create Instance or the Edit Properties commands. If you modify the parameter values through other commands then the CDF parameter callback is not invoked.

You can enter a callback as a SKILL expression that you type directly in the callback field of the parameter section of the Edit CDF form. You can also enter a callback by using the *?callback* keyword as an argument to the *cdfCreateParam* function:

```
cdfParamId = cdfCreateParam(cdfDataId
?type "string"
?parseAsNumber "yes"
?units "lengthMetric"
?parseAsCEL "yes"
?storeDefault "no"
?name "pName2"
?prompt "width"
?defValue 7u
?display "artParameterInToolDisplay('w)"
?editable "cdfgData->maskLayoutViewName->..."
?callback "cdfgData->pName1->value =
    cdfgData->pName2->value")
```

In this example, the callback on parameter *pName2* sets the value of parameter *pName1* to the new value when the value of *pName2* changes.

A callback can also be a call to a function that you loaded previously.

```
cdfParamId = cdfCreateParam(cdfDataId
?type "string"
?parseAsNumber "yes"
?units "lengthMetric"
?parseAsCEL "yes"
?storeDefault "no"
?name "pName2"
?prompt "width"
?defValue 7u
```

```
?display "artParameterInToolDisplay('w)"
?editable "cdfgData->maskLayoutViewName->..."
?callback "myFunc( )")
```

In the second example, the callback on parameter *pName2* calls the function *myFunc( )* when the value of *pName2* changes.

When using callbacks, always use the global variable *cdfgData* to access information about parameter values and default values.

To get the default value for a parameter, type the following expression:

```
cdfgData->paramName->defValue
```

To get the current value for a parameter, type the following expression:

```
cdfgData->paramName->value
```

To set the value of a parameter, type either of the following expressions:

```
cdfgData->paramName->value = 10
cdfgData->paramName->value =
    cdfgData->paramName->defValue
```

Loading Callbacks

Once you define the SKILL callback procedures, you need to load the files. From the Command Interpreter Window in your `.cdsinit` file, type

```
load("path/callbacks.il")
```

You must make sure that the callbacks are loaded before you use the library and that the callback files are archived whenever you archive the library.

As an alternative, you can attach a callback file to a library. You attach callback files to a library by including a `libInit.il` file in that library. `libInit.il` is a text file that contains calls to load other SKILL files or contexts in the library or to initialize autoload properties.

You must always use the file name `libInit.il` or `libInit.ils`. You can have one such file in each library. The following example is the contents of a `libInit.il` file in a microwave library:

```
load( ddGetObj( "microwave" nil nil "libraProcs" )~>path )
load( ddGetObj( "microwave" nil nil "hpmnsProcs" )~>path )
load( ddGetObj( "microwave" nil nil "mharmProcs" )~>path )
load( ddGetObj( "microwave" nil nil "callbacks" )~>path )
```

where `ddGetObj` uses the following syntax:

```
ddGetObj( libName cellName viewName fileName)
```

If you attach the file to the library, the file is automatically loaded whenever the library is opened. This guarantees that the functions are defined. In addition, when you archive your library, the CDF callback functions are automatically archived.

NFET Example

To understand how to use callbacks, look at the NFET CDF description. The following callback functions are specified for the NFET component:

- *mosLayout*( ) on parameter *maskLayoutViewName*
- *pcMOSw*( ) on parameter *w*
- *pcMOSl*( ) on parameter *l*

mosLayout Callback

In the NFET CDF description, when you select the *layout.pwr* option of the *maskLayoutViewName* parameter, the values of the other parameters are constant. You can use the *mosLayout* callback procedure to set these values automatically whenever the *maskLayoutViewName* is *layout.pwr*. As the following example shows, whenever the value of *maskLayoutViewName* changes, the callback checks the value of *maskLayoutViewName* and, if the value is *layout.pwr*, sets the parameter values:

```
procedure( mosLayout( )
    if((cdfgData->maskLayoutViewName->value ==
        cdfgData->maskLayoutViewName->defValue)
    then      ; It is layout. Use defaults.
        cdfgData->w->value=cdfgData->w->defValue
        cdfgData->width->value=cdfgData->width->defValue
        cdfgData->l->value=cdfgData->l->defValue
        cdfgData->length->value=cdfgData->length->defValue
        cdfgData->bend->value=cdfgData->bend->defValue
        cdfgData->bendt->value=cdfgData->bendt->defValue
        cdfgData->ad->value=cdfgData->ad->defValue
        cdfgData->as->value=cdfgData->as->defValue
        cdfgData->pd->value=cdfgData->pd->defValue
        cdfgData->ps->value=cdfgData->ps->defValue
        cdfgData->diffD->value=cdfgData->diffD->defValue
    else      ; It is layout.pwr. Use known values.
        cdfgData->w->value="880u"
        cdfgData->width->value=880
        cdfgData->l->value="12u"
        cdfgData->length->value=12
        cdfgData->bend->value=cdfgData->bend->defValue
        cdfgData->bendt->value=cdfgData->bendt->defValue
        cdfgData->ad->value="6.16e-09"
        cdfgData->as->value="6.16e-09"
        cdfgData->pd->value="0.001774"
        cdfgData->ps->value="0.001774"
        cdfgData->diffD->value=nil
    )
)
```

pcMOSw Callback

The *pcMOSw* callback is similar to the *pcMOSl* callback*. pcMOSw* checks the input, sets the *width* parameter, determines if there should be a bend, and calculates some of the simulation parameters:

```
procedure( pcMOSw( )
  prog(()
    if( (cdfgData->maskLayoutViewName->value ==
            cdfgData->maskLayoutViewName->defValue)
                 ; Do this only if maskLayoutViewName is
                 ; layout. Otherwise, mosLayout sets it.
    then
        tmp = evalstring(cdfgData->w->value)
        if( (tmp && (typep(tmp)!='flonum))
        then
            artError("Width value must be a floating
                 point number. Set to default." )
            cdfgData->w->value = cdfgData->w->defValue
            cdfgData->width->value =
                 cdfgData->width->defValue
            cdfgData->bend->value =
                 cdfgData->bend->defValue
            cdfgData->bendt->value =
                 cdfgData->bendt->defValue
            cdfgData->ad->value = cdfgData->ad->defValue
            cdfgData->as->value = cdfgData->as->defValue
            cdfgData->pd->value = cdfgData->pd->defValue
            cdfgData->ps->value = cdfgData->ps->defValue
            return(nil)
        )
        tmp = ( tmp / 1e-6 )
        if( (tmp < cdfgData->width->defValue)
        then
            artError("Width value is less than
                 minimum (7u). Set to default." )
            cdfgData->w->value = cdfgData->w->defValue
            cdfgData->width->value =
                 cdfgData->width->defValue
            cdfgData->bend->value =
                 cdfgData->bend->defValue
            cdfgData->bendt->value =
                 cdfgData->bendt->defValue
            cdfgData->ad->value = cdfgData->ad->defValue
            cdfgData->as->value = cdfgData->as->defValue
            cdfgData->pd->value = cdfgData->pd->defValue
            cdfgData->ps->value = cdfgData->ps->defValue
            return(nil)
        )
        if( (tmp > 240 )
        then
            artError("Width value is greater than
                 maximum (240u). Set to default." )
            cdfgData->w->value = cdfgData->w->defValue
            cdfgData->width->value =
                 cdfgData->width->defValue
            cdfgData->bend->value =
                 cdfgData->bend->defValue
            cdfgData->bendt->value =
```

```
                cdfgData->bendt->defValue
            cdfgData->ad->value = cdfgData->ad->defValue
            cdfgData->as->value = cdfgData->as->defValue
            cdfgData->pd->value = cdfgData->pd->defValue
            cdfgData->ps->value = cdfgData->ps->defValue
            return(nil)
        )
        if( (tmp <= 240 )
        then
            cdfgData->maskLayoutViewName->value =
                cdfgData->maskLayoutViewName->defValue
            grid = round(tmp/0.5)
            newwidth = grid*0.50
            if( (newwidth != tmp)
            then
                artError("Width is set to nearest value
                    on 0.50 micron pitch.")
                cdfgData->w->value =
                    sprintf(s "%g" (newwidth * 1e-6))
            )
            if( (tmp > 120) ;criteria for setting bend here
            then
                cdfgData->width->value = tmp/2
                cdfgData->bend->value = t
                cdfgData->bendt->value = 1
                cdfgData->diffD->value = nil
            else
                cdfgData->width->value = tmp
                cdfgData->bend->value =
                    cdfgData->bend->defValue
                cdfgData->bendt->value =
                    cdfgData->bendt->defValue
            )
        )
        cdfgData->ad->value =
            sprintf(s "%g" (tmp * 7.0 * 1e-12))
        cdfgData->as->value =
            sprintf(s "%g" (tmp * 7.0 * 1e-12))
        cdfgData->pd->value =
            sprintf(s "%g" ((tmp + 7.0) * 2e-6))
        cdfgData->ps->value =
            sprintf(s "%g" ((tmp + 7.0) * 2e-6))
    )
  )
)
```

pcMOSl Callback

When the *length* and *width* parameters for the NFET CDF description were created, the *l* and *w* parameters already existed for one of the simulators. Instead of prompting the user twice for the same information, callbacks attached to the *l* and *w* parameters can update the *length* and *width* parameters automatically.

The *pcMOSl* callback runs whenever the value of *l* changes. The callback verifies that the *maskLayoutViewName* parameter value is *layout*. (For the other option, *layout.pwr*, the parameter values are controlled by the *mosLayout* callback, shown in the previous section.) The callback then checks the new value against the minimum value, the maximum value, and the grid. If the value is not valid, the callback resets the lengths to the default value. If the value is valid, the callback sets the *length* parameter value. The following example shows this process:

```
procedure( pcMOSl( )
  prog(())
    if((cdfgData->maskLayoutViewName->value ==
        cdfgData->maskLayoutViewName->defValue)
                 ; Do this only if maskLayoutViewName is
                 ; layout. Otherwise, mosLayout sets it.
    then
                 ; Convert input string to float.
        tmp = evalstring(cdfgData->l->value)
        if( (tmp && (typep(tmp)!='flonum))
        then
            artError("Length value must be a floating point
                 number. Set to default." )
                 ; artError will send message to CIW
            cdfgData->l->value = cdfgData->l->defValue
            cdfgData->length->value =
            cdfgData->length->defValue
            return(nil)
        )
        tmp = ( tmp / 1e-6 )    ; Convert to user units.
        if( (tmp < 3.0)
        then
            artError("Length is less than minimum (3u).
                 Set it to default." )
            cdfgData->l->value = cdfgData->l->defValue
            cdfgData->length->value =
                 cdfgData->length->defValue
            return(nil)
        )
        if( (tmp > 15.0 )
        then
            artError("Length value is greater than
                 maximum (15u). Set it to default." )
            cdfgData->l->value=cdfgData->l->defValue
            cdfgData->length->value =
                 cdfgData->length->defValue
            return(nil)
        )
                 ; The following checks the value against
                 ; the grid and fixes it if necessary.
```

```
        grid = round(tmp/0.5)
        newlength = grid*0.50
        if( (newlength != tmp)
        then
            or("Length is set to nearest value
                 on 0.50 micron pitch.")
            cdfgData->l->value = sprintf(s "%g"
                 (newlength * 1e-6))
        )
                 ; Set the pcell param value.
        cdfgData->length->value = tmp
    )
  )
)
```

8

CDF SKILL Summary

Overview

Short examples in the previous chapters describe how to use the Cadence® SKILL language for specific operations. This chapter:

- Covers all CDF SKILL functions in general
- Describes each type of element in a CDF SKILL function
- Summarizes all the CDF SKILL functions

When you use these SKILL functions, you can perform all the same operations that you do when you fill in form fields and apply them. However, unlike filling out forms, with SKILL you can create routines to perform the same operation on many components at once, automating your handling of CDF descriptions.

CDF SKILL Function Elements

CDF SKILL functions use or operate on data IDs, parameters, parameter attributes, expressions, and global variables. This section describes each of these elements in the context of SKILL.

Cell and Library Data IDs

Before working on a CDF description, you must specify the data ID of the cell or library. You can get the data ID for the cell or library you are using with the `ddGetObj()` function. With SKILL versions earlier than 4.4, you used the `dmFindLib()` and `dmFindCell()` functions. You must assign the value returned by these functions to a variable that you create. For example, to operate on the *analogLib* library, you must first create the variable *mylib* with this assignment:

```
mylib = ddGetObj("analogLib")
```

When creating the variable for the object ID of a cell, you must specify both the library name and the cell name:

```
test_cell = ddGetObj("analogLib" "schottky")
```

or

```
test_cell = ddGetObj(mylib "schottky")
```

if you have already defined *mylib*.

You must then use *mylib* when a SKILL function requires a *d_id* or *d_libId*, and test_cell when a SKILL function requires a *d_id* or *d_cellId*.

You can use this data ID to access information about the description by using the right arrow (-> or ~>) operator. For example

```
test_cell -> name
```

returns the ID

```
"schottky"
```

Data Objects

CDF descriptions are represented by *cdfDataId* objects, which are SKILL objects that you can manipulate like other SKILL database objects. Just like data IDs, you assign the *cdfDataId* of a particular CDF description to a variable that you create. The most common variable name is *cdfDataId*. To create a new CDF data object for a new CDF description, use this type of ID assignment:

```
newCellId = cdfCreateUserCellCDF(test_cell)
cdf:25092160
newCellId->type
"userCellData"
```

To access an existing CDF data object for an existing CDF description, use this type of ID assignment:

```
baseCell = cdfGetBaseCellCDF(test_cell)
baseCell->dataFile->value
"bjt"
```

newCellId and *baseCell* are arbitrary names that you create.

Although you can add any information to a *cdfDataId*, the object has specific fields to hold the information that it maintains. These fields are described in the following section.

id

The database object (cell ID or library ID) to which the *cdfDataId* is attached. This field is not editable.

type

There are seven types of *cdfDataId*:

baseLibCDF	base-level library CDF
userLibCDF	user-level library CDF
effLibCDF	effective-level library CDF
baseCellCDF	base-level cell CDF
userCellCDF	user-level cell CDF
effCellCDF	effective-level cell CDF
effInstCDF	effective-level instance CDF

This field is not editable.

parameters

List of parameters attached to this *cdfDataId*. This field is not editable.

doneProc

An optional procedure name (a string) that is evaluated after any change to a parameter on a component instance. You can use *doneProc* for post-processing. The procedure must take a single argument: the instance that has been modified.

The default value is `nil`. This field is editable.

formInitProc

An optional procedure name. If specified, the procedure is executed when the contents of the CDF are displayed on a form. The default value is *nil*. This field is editable.

This procedure runs when you use the *Add Instance* and *Edit Properties* commands. You can use the procedure for preprocessing CDF data. This procedure must take a single argument, the *cdfDataId* being added to the form.

Note: When you modify a parameter field value using the *formInitProc*, the Edit Properties Object command for the schematic application is not aware of the modification and does not update the changes made with *formInitProc*. You can avoid this problem by setting the variable *cdfgForm->cdfModified* to the value *t*.

Displaying Parameters

The following fields control the display of parameters on a form.

fieldWidth	Width of a field	Default = 350 pixels
fieldHeight	Height of a field	Default = 35 pixels
buttonFieldWidth	Width of a button field	Default = 340 pixels
promptWidth	Width of a prompt	Default = 175 pixels

Parameters

CDF parameters are represented by *cdfParamId* objects, which are SKILL objects that you can manipulate like other SKILL database objects. You can use a *cdfParamId* to access information about a parameter by using the right arrow (->) operator. Although you can add any information to a *cdfParamId*, the object has fields to hold information that it maintains.

In addition to getting the *cdfParamId*s through the parameters field of a *cdfDataId*, you can also access a parameter by specifying:

`cdfDataId->paramName`

`paramName` is the name of the parameter. However, you must not create any user-defined properties on a *cdfDataId* that conflict with the name of a parameter on the *cdfDataId*.

use

Specifies whether to use this parameter. The *use* attribute is context-specific and is evaluated when necessary. In this field, you can specify parameters that you are not using because of the value of other parameters, or because of the function you are using. The *use* field must be a string.

- If the field evaluates to *non-nil*, the system uses the parameter.
- If a value for the field exists, the system ignores the value.
- If the field evaluates to *nil*, the system does not use the parameter.
- If you do not specify this field, *t* is assumed, so the system always uses the field.

The “Global Variables” section describes several global variables for constructing the *use* expression.

This field is editable.

paramType

Type of the parameter. Valid values for this field are

```
"string"
"int"
"float"
"radio"
"cyclic"
"boolean"
"button"
"netSet"
```

The quotation marks are required. CDF checks that the parameter value matches the type specified. You must specify this field.

This field is not editable.

Note: Use the netSet type to store inherited connections in CDF. Inherited connections allow you to selectively override global signals in designs created in Virtuoso Schematic Editor. The override information is communicated through net expressions and netSet properties. For more information on netSet properties refer to *Inherited Connections Flow Guide* and *Virtuoso Schematic Editor L User Guide*.

defValue

Default value of the parameter. You must specify this field.

This field is editable.

value

Current value of the parameter.

This field is editable.

prompt

Prompt that appears on a form field when an application asks for the value of this parameter. The value for this field must be a string. You must specify this field.

This field is editable.

choices

Possible values (a list of strings) of a *cyclic* or *radio* type parameter. Do not use this field for other types of parameters.

This field is editable.

units

Unit type of the parameter. You often modify parameters that represent resistance, capacitance, length, time, power, and so forth, and expect to see the parameter value scaled to appropriate units (for example, *pF*, *uM*, *dBm*). Valid values for this field include the following strings:

```
"resistance"
```

```
"capacitance"
"inductance"
"conductance"
"time"
"frequency"
"power"
"powerDB"
"lengthMetric"
"lengthEnglish"
"angle"
"voltage"
"current"
"temperature"
```

The quotation marks are required. This field determines the unit's suffix and scale factor to use when displaying the parameter value. For example, by setting a parameter's units field to *capacitance,* the parameter value is displayed as *5 pF* instead of *5e-12*.

editable

Specifies whether a parameter is for display only. If set to *nil*, the field specifies a displayed parameter that you cannot change. (Other parameters' callbacks can change the parameter.) Use this field only on *int*, *float*, and *string* type fields.

The *editable* field must be a string. This field is evaluated when necessary.

- If the field evaluates to *non-nil*, the parameter is editable.
- If the field evaluates to *nil*, the parameter is grayed out when displayed as a form field.
- If you do not specify the field, *t* is assumed, meaning that the field is editable.

callback

SKILL code to be executed whenever the value of a parameter changes. Using *callback*, you can cause any parameter's value to affect any other parameter's value. The value of this field must be a string.

This field is optional. If you do not use *callback*, no callback is executed.

The “Global Variables” section describes several global variables you can use in the *callback* field.

parseAsNumber

String type parameter whose value can evaluate to a floating-point number. These types of parameters often occur for circuit level simulators that allow component parameters to be either numbers or expressions containing variables.

If you specify this field, the system parses the value of the parameter to check if the value is a number.

- If the value is a number, the system converts the string to a floating point number, converts the string to the most efficient notation (taking into account any units field specified), then reconverts the number into a string.
- If the parameter value is not a number, the system does no conversion.

For example, if the parameter value is the string *5.4e-12* and the parameter value has a units field of *capacitance* (that is, the suffix is F), the parameter value is converted to the string *5.4p* before being displayed. If, however, the parameter value is *c1*, no conversion takes place.

dontSave

Specifies that the parameter cannot be stored in any instance that corresponds to the component.

The *dontSave* field must be a string. The system evaluates the field as necessary. If the field evaluates to non-*nil*, the parameter is not stored. If you do not specify the field, *nil* is assumed and the parameter value is stored.

parseAsCEL

If set to *yes*, the associated CDF parameter is processed as a CDF Expression Language (CEL) expression. The parameter must be a string. If the expression resolves to a numeric value (the usual case), the *parseAsNumber* flag should also be set to *yes*.

storeDefault

Specifies whether the parameter default value is saved as a property of the instance. All tools based on the Cadence Analog Design Environment software use the CDF to find default values if no property exists on an instance.

If set to *no* (the default) or *don't use*, a property is not saved on the instance when the parameter value is the default. Also, if the default value of the parameter changes, all instances that use the default automatically get the new default value. (To see the change in an open window, you must select *Window – Redraw* from the Cadence menu.)

If set to *yes* and the parameter value is set to the same value as the default, a property is saved on the instance. One disadvantage of this attribute is that if the default value of a parameter changes, the instances that use the default do not automatically change to the new default. If you are a user of the Open Simulation Software (OSS) system, and you used to set this attribute to *yes* because you could only netlist using instance properties, you can now set it to *no* because OSS users now have the option of using CDF.

display

Determines if this parameter is displayed in forms that display CDF parameters, such as the Edit Object Properties form or the Add Instance form. You must enter `t`, `nil`, or a SKILL expression that evaluates to `t` or `nil` in this field to determine if this parameter is to be displayed. If the field evaluates to non-`nil` (the default), the parameter is displayed. If the field evaluates to `nil`, the parameter is not displayed.

Example

You can use the *cdfDataId baseCell* from the previous example to examine the parameters in the base-level cell CDF description of the Schottky transistor in the analog library. For example, that cell has a parameter *m*. You can access information about *m* in the following manner:

```
baseCell->m->prompt
"Multiplier"
baseCell->m->editable
nil
baseCell->m->display
"artParameterInToolDisplay('m)"
baseCell->m->paramType
"string"
```

Expressions

CEL (CDF Expression Language) is another name for the Analog Expression Language (AEL) that works with CDF parameters. For information on AEL, refer to the *Analog Expression Language Reference*. With AEL you can express a value as a mathematical

expression instead of a single, fixed value. In such an expression, symbolic names such as *sheetResistivity* can refer to values that are computed and set on one of the following:

- CDF parameter on the same design component
- CDF parameter on the parent design component

Global Variables

CDF parameter values interact with each other, and one parameter's value can affect the existence of another. This feature is implemented primarily through the *use* and *callback* fields of the parameters section. The following global variables, which you can access, are set whenever parameter fields are evaluated.

cdfgData

CDF data for the component in use. Use the *value* field to get the current value of any parameter in the CDF description.

- For creating an instance, set the field to the last value you used when you created a component of this type.
- For editing, set the field to the value for the component you are editing.

For example, you might set the *use* field for the *resistance* parameter to

```
"cdfgData->resType->value == \"ideal\""
```

implying that the *resistance* parameter should be used only if the resistor type is set to *ideal*. (== is an equality test, not an assignment.)

When setting the value of a parameter, set

```
cdfgData->paramName->value = paramValue
```

If you use this setting, in the future you should be able to use the property list editor to edit the parameter values directly without going through the form.

cdfgForm

Form on which the CDF data is displayed, if there is one. You can modify data stored in *cdfgData* in the CDF *formInitProc*, or by modifying the *cdfgForm* fields with a callback function. When doing this, set the Boolean variable *cdfgForm -> cdfModified* to *t*.

gLabelsNumNotation

Displays the cdsTerm and cdsParam values in different notations, such as scientific or engineering. The syntax of `gLabelsNumNotation` is as follows:

```
gLabelsNumNotation = 'suffix
```

The possible values can be `'suffix`, `'scientific`, `'engineering`, or `'default`.

The `default` value displays the labels according to the existing setting. For example, the nmos symbol shows the engineering notation, `300e-3` for `w`, by default.

To set the number of significant digits, use the `aelPushSignifDigits` function as follows:

```
aelPushSignifDigits(10)
```

CDF SKILL Functions

This section describes the functions that operate on CDF descriptions.

Create Functions

You can create base and user CDF descriptions for libraries and cells using the following functions.

cdfCreateBaseLibCDF

```
cdfCreateBaseLibCDF(
     libId
     [?doneProc t_doneProc]
     [?formInitProc t_formInitProc]
     [?fieldWidth x_fieldWidth]
     [?fieldHeight x_fieldHeight]
     [?buttonFieldWidth x_buttonFieldWidth]
     [?promptWidth x_promptWidth]
     )
     => cdfDataId / nil
```

Description

Creates the Base Library CDF that is applied to all the devices in the library. The CDF description is created with no parameters or *simModels*.

Note the following:

- You must open the library in write mode.
- Before using this function, ensure that the base-level CDF description does not already exist for the library. If the CDF description already exists, this function will not update the existing CDF description.

Arguments

`libId`	This is the library ID.
`t_doneProc`	Lets you specify an optional SKILL routine that executes after you change any parameter on the instantiation form.

`t_formInitProc`	Lets you specify an optional SKILL language routine that executes automatically when the component is placed on an instantiation form.
`x_fieldWidth`	Lets you specify the width of a field on the instantiation form. The default width is 350 pixels.
`x_fieldHeight`	Lets you specify the height of a field on the instantiation form. The default height is 35 pixels.
`x_buttonFieldWidth`	Lets you specify the width of a button on the instantiation form. The default width is 350 pixels.
`x_promptWidth`	Lets you specify the width of the prompt on the instantiation form. The default width is 175 pixels.

cdfCreateUserLibCDF

```
cdfCreateUserLibCDF(
     libId
     [?doneProc t_doneProc]
     [?formInitProc t_formInitProc]
     [?fieldWidth x_fieldWidth]
     [?fieldHeight x_fieldHeight]
     [?buttonFieldWidth x_buttonFieldWidth]
     [?promptWidth x_promptWidth]
     )
     => cdfDataId / nil
```

Description

Creates the user-level library CDF that is applied to all the devices in the library. The user-level CDF can override entries in the base-level CDF. Therefore, a combination of the base-level CDF and the user-level CDF becomes the effective CDF.

The CDF description is created with no parameters or simulation models.

Note: Before using this function, ensure that the user-level CDF description does not already exist for the library. If the CDF description already exists, this function will not update the existing CDF description.

Arguments

`libId`	This is the library ID.

`t_doneProc`	Lets you specify an optional SKILL routine that executes after you change any parameter on the instantiation form.
`t_formInitProc`	Lets you specify an optional SKILL language routine that executes automatically when the component is placed on an instantiation form.
`x_fieldWidth`	Lets you specify the width of a field on the instantiation form. The default width is 350 pixels.
`x_fieldHeight`	Lets you specify the height of a field on the instantiation form. The default height is 35 pixels.
`x_buttonFieldWidth`	Lets you specify the width of a button on the instantiation form. The default width is 350 pixels.
`x_promptWidth`	Lets you specify the width of the prompt on the instantiation form. The default width is 175 pixels.

cdfCreateBaseCellCDF

```
cdfCreateBaseCellCDF(
     cellId
     [?doneProc t_doneProc]
     [?formInitProc t_formInitProc]
     [?fieldWidth x_fieldWidth]
     [?fieldHeight x_fieldHeight]
     [?buttonFieldWidth x_buttonFieldWidth]
     [?promptWidth x_promptWidth]
     )
     => cdfDataId / nil
```

Description

Creates a base-level CDF description for a cell. The CDF description is created with no parameters or simulation models.

Note the following:

- You must open the cell in write mode.
- Before using this function, ensure that the base-level CDF description does not already exist for the cell. If the CDF description already exists, this function will not update the existing CDF description.

Arguments

`cellId`	This is the cell ID.
`t_doneProc`	Lets you specify an optional SKILL routine that executes after you change any parameter on the instantiation form.
`t_formInitProc`	Lets you specify an optional SKILL language routine that executes automatically when the component is placed on an instantiation form.
`x_fieldWidth`	Lets you specify the width of a field on the instantiation form. The default width is 350 pixels.
`x_fieldHeight`	Lets you specify the height of a field on the instantiation form. The default height is 35 pixels.
`x_buttonFieldWidth`	Lets you specify the width of a button on the instantiation form. The default width is 350 pixels.
`x_promptWidth`	Lets you specify the width of the prompt on the instantiation form. The default width is 175 pixels.

cdfCreateUserCellCDF

```
cdfCreateUserCellCDF(
     cellId
     [?doneProc t_doneProc]
     [?formInitProc t_formInitProc]
     [?fieldWidth x_fieldWidth]
     [?fieldHeight x_fieldHeight]
     [?buttonFieldWidth x_buttonFieldWidth]
     [?promptWidth x_promptWidth]
     )
     => cdfDataId / nil
```

Description

Creates a user-level CDF description for a cell. The CDF description is created with no parameters or simulation models.

Note: Before using this function, ensure that the user-level CDF description does not already exist for the cell. If the CDF description already exists, this function will not update the existing CDF description.

Arguments

`cellId`	This is the cell ID.
`t_doneProc`	Lets you specify an optional SKILL routine that executes after you change any parameter on the instantiation form.
`t_formInitProc`	Lets you specify an optional SKILL language routine that executes automatically when the component is placed on an instantiation form.
`x_fieldWidth`	Lets you specify the width of a field on the instantiation form. The default width is 350 pixels.
`x_fieldHeight`	Lets you specify the height of a field on the instantiation form. The default height is 35 pixels.
`x_buttonFieldWidth`	Lets you specify the width of a button on the instantiation form. The default width is 350 pixels.
`x_promptWidth`	Lets you specify the width of the prompt on the instantiation form. The default width is 175 pixels.

cdfCreateParam

```
cdfCreateParam(g_cdfDataId ?name t_name ?type t_type [?defValue g_defValue]
     [?units t_units]
     [?parseAsNumber t_parseAsNumber]
     [?choices l_choices]
     [?prompt t_prompt]
     [?use t_use]
     [?display t_display]
     [?editable t_editable]
     [?dontSave t_dontSave]
     [?callback t_callback]
     [?storeDefault t_storeDefault])
     [?parseAsCEL t_parseAsCEL]
     )
     => cdfDataId / nil
```

Description

Creates a parameter on the specified `cdfDataId` with the specified attributes. The only attributes that are *always* required are the parameter's name and type. If this parameter description is not overriding an existing base-level parameter definition, you must also specify the *defValue*.

Note the following:

- You cannot override a parameter's type. Also, CDF checks that the effective parameter due to any overrides is consistent and valid.
- If the name of a parameter you are adding using this function is the same as the name of a parameter that already exists in the CDF description, the CDF description will not be updated for that parameter.

Note: Refer to Component Parameters on page 59 in Chapter 3 for considering different options during the parameter creation.

Example

```
cdfParamId = cdfCreateParam(cdfDataId
?name "resistorType"
?type "radio"
?prompt "Type:"
?defValue "ideal"
?choices list("ideal" "thin film")
?callback "myModelTypeCB()")
```

For more examples of the `cdfCreateParam` function, refer to Appendix C, "NBSIM Transistor CDF SKILL Description".

Queries

You can query existing CDF descriptions using the following functions. To use the returned value, you must assign it to a variable that you create.

cdfGetBaseLibCDF

```
cdfGetBaseLibCDF(d_libId
     )
     => cdfDataId / nil
```

Returns the base-level CDF description attached to a library. If one is not defined, it returns `nil`.

cdfGetUserLibCDF

```
cdfGetUserLibCDF(d_libId
     )
     => cdfDataId / nil
```

Returns the user-level CDF description attached to a library. If one is not defined, it returns `nil`.

cdfGetLibCDF

```
cdfGetLibCDF(d_libId
     )
     => cdfDataId / nil
```

Returns the effective CDF description attached to a library. If neither a base- nor user-level CDF description is defined, it returns `nil`. The resulting CDF description represents the overlay of the user-level CDF on the base-level CDF.

cdfGetBaseCellCDF

```
cdfGetBaseCellCDF(d_cellId
     )
     => cdfDataId / nil
```

Returns the base-level CDF description attached to a cell. If one is not defined, it returns `nil`.

cdfGetUserCellCDF

```
cdfGetUserCellCDF(d_cellId
     )
     => cdfDataId / nil
```

Returns the user-level CDF description attached to a cell. If one is not defined, it returns `nil` .

cdfGetCellCDF

```
cdfGetCellCDF(d_cellId
     )
     => cdfDataId / nil
```

Returns the effective CDF description attached to a cell. If neither a base- nor user-level CDF description is defined for the cell or its library, it returns `nil`. The resulting CDF description represents the overlay of the user-level cell CDF on the base-level cell CDF on the user-level library CDF on the base-level library CDF.

cdfGetInstCDF

```
cdfGetInstCDF(d_instId
     )
     => cdfDataId / nil
```

Returns the effective CDF description associated with an instance.

The difference between the instance's effective CDF description and the cell's effective CDF description is that the values of any CDF parameter takes into account the values of the parameters stored on the instance.

Saving Descriptions

You can save CDF descriptions using the `cdfSaveCDF` function. If this CDF description already exists, the new description is written over the old one.

cdfSaveCDF

```
cdfSaveCDF(g_cdfDataId
     )
     => t / nil
```

Saves a CDF description to disk.

The CDF description is then read in every time you open the cell of the library to which the description is attached. You can save only base-level CDF descriptions. You must have write permission on the object to which the CDF description is attached to execute this function.

Dumping and Editing Descriptions

You can save a CDF description to a file and edit it using the following functions. These functions replace `adcDump` and `cdfDumpCellCDF`.

Caution

While editing dump files remember to escape invalid characters while specifying termOrder using symbols. All symbols must be preceded by the backslash (\) to make them valid symbols in SKILL.

cdfDump

```
cdfDump(t_libName t_fileName
     [?cellName t_cellName]
     [?level s_level]
     [?edit b_edit]
     )
     => t / nil
```

Dumps the CDF description for *`t_libName`* and *`t_cellName`* into *`t_fileName`*. If *`t_cellName`* is not specified, then only the library CDF description is dumped. *`t_fileName`* is created in the current working directory or the directory specified with the filename. *`s_level`* is either `'base` or `'user`, with `'base` as the default value. If *`b_edit`* *is* `t`, a text editor window is automatically opened on *`t_fileName`*. The default is no editor.

Example

```
cdfDump( "nmos" "tr.mod" ?cellName "pnp" ?level 'base
    ?edit t)
```

cdfDumpAll

```
cdfDumpAll(t_libName t_fileName
     [?level s_level]
     [?edit b_edit]
     )
     => t / nil
```

Dumps the CDF description for *`t_libName`* and all its cells into *`t_fileName`*. *`s_level`* is either `'base` or `'user`, with `'base` as the default value. *`t_fileName`* is created in the current working directory or the directory specified with the filename. If *`b_edit`* *is* `t`, a text editor window is automatically opened on *`t_fileName`*. The default is no editor.

Example

```
cdfDumpAll("asic" "lib.mod" ?level 'base ?edit t)
```

Deleting Descriptions

You can delete a CDF description or its parameters using the following functions.

cdfDeleteCDF

```
cdfDeleteCDF(g_cdfDataId
     )
     => t / nil
```

Deletes a CDF description, including all attached parameters.

If the CDF description has been saved, the saved versions are also deleted. If this is a base-level CDF description, you must have write permission on the object to which the CDF description is attached.

cdfDeleteParam

```
cdfDeleteParam(g_cdfParamId
     )
     => t / nil
```

Deletes a CDF parameter.

If the CDF parameter is attached to a base-level CDF description, you must have write permission on the object to which the CDF description is attached.

CDF checks that no invalid parameter descriptions would result from deleting the specified parameter before it actually deletes it. This would occur if you tried to delete a base-level parameter and a user-level parameter is defined that only partially overrides the base-level description.

Copying, Finding, and Updating Data and Parameters

You can copy, find, and update CDF data and parameters using the following functions.

cdfCopyCDF

```
cdfCopyCDF(d_id t_dataType g_dataId
     )
     => g_cdfDataId / nil
```

Copies CDF data onto the specified ID, creating a new *dataId* of the specified type.

The ID must be either a library or a cell, and it must not already have a CDF description of the specified type. The data type must be one of the following:

```
"baseLibData"
"userLibData"
"baseCellData"
"userCellData"
```

Neither the old nor the new *dataId* can be effective objects.

Arguments

d_id	Specifies the ID of the library or cell
t_dataType	Specifies the type of cdf data to be copied
g_cdfDataId	Specifies the source of the cdf data to be copied from

Value Returned

cdfDataId	New cdf data returned if successful
nil	nil returned in case of an error

cdfCopyParam

```
cdfCopyParam(g_dataId g_paramId
     )
     => g_paramId / nil
```

Copies a parameter, adding it to *dataId*.

dataId must not already have a parameter with the same name. The parameter and the dataId must not be effective objects.

cdfFindParamByName

```
cdfFindParamByName(g_cdfDataId t_name
     )
     => g_paramId / nil
```

Returns the parameter ID for the specified parameter name on the specified CDF description, if it exists. If not, it returns `nil`.

Use this function to search for parameters by name.

cdfUpdateInstParam

```
cdfUpdateInstParam(
     d_instId
     )
     => t / nil
```

Stores the CDF parameters specified in the effective cell CDF of the instance master onto the specified instance. When the Id given is not for an instance or the instance master does not have CDF definition, it returns `nil`.

cdfRefreshCDF

```
cdfRefreshCDF(
     d_libId/d_cellId
     )
     => t / nil
```

Updates the CDF structure in the memory for the specified library or cell Id with the contents stored on the hard disk. Returns `nil` if the CDF structures for the specified library and cell Id is not present in memory.

Caution

Use this function with caution. In case multiple users modify the CDF parameters of a common component simultaneously, the saved parameter pointers may become invalid as there is no way of notifying the application after refreshing the CDF parameter values.

Arguments

`d_libId or d_cellId`	Specify the *`dd_id`* of the library or the cell.

Example

```
cdfRefreshCDF(ddGetObj("mylib" "npnpar"))
```

aedCopyCDF

```
aedCopyCDF(
     )
     => t / nil
```

Opens the Copy Component CDF form.

aedDeleteCDF

```
aedDeleteCDF(
     )
     => t / nil
```

Opens the Delete Component CDF form.

Setting Scale Factors

Use the following Cadence SKILL language commands to determine the current scale factors or to set scale factors.

cdfGetUnitScaleFactor

```
cdfGetUnitScaleFactor(
     t_unitName
     )
     => t_scaleFactor
```

Description

Displays the current scale factor for the specified unit.

Arguments

`t_unitName`	The unit name for which you want to display the scale factor.

Value Returned

`t_scaleFactor`	The current scale factor for the specified unit name.

Example

Following command returns the scale factors for *power*:

```
cdfGetUnitScaleFactor("power")
```

cdfSetUnitScaleFactor

```
cdfSetUnitScaleFactor(
     t_unitName
     t_scaleFactor
     )
     => t / nil
```

Description

Sets the scale factor for the specified unit.

Arguments

`t_unitName`	The unit name for which you want to set the scale factor.
`t_scaleFactor`	The scale factor for the specified unit name.

Example

Following command sets the lengthMetric to m (millimeters):

```
cdfSetUnitScaleFactor("lengthMetric" "m")
```

Other SKILL Functions

cdfParseFloatString

```
cdfParseFloatString(
     t_string
     )
     => nil / d_value / t_string
```

Description

This function uses the standard `strtod` (string to double) function to parse the input string. When the input string contains trailing non-numerical characters, the fragment of the string is compared against a supported set of scale factor designators.

For information on scale factors, refer to Chapter 2, “CDF Commands.”

Value Returned

nil	when the input string cannot be parsed as a float value or without a valid scale factor, as shown below: `cdfParseFloatString("1g") => nil`

`d_value` (a float value)	when the input string can be parsed as a float value with or without a valid scale factor, as shown below:
	`cdfParseFloatString("1.0") => 1.0`
	`cdfParseFloatString("1.0u") => 1e-06`

the given string	when the input string does not contain a valid numerical representation for a float value. For example, the input string starting with a non-digit character as shown below: `cdfParseFloatString("abcd") => "abcd"`

cdfFormatFloatString

```
cdfFormatFloatString(
     t_string
     t_scaleFactor
     )
     => nil / t_val
```

Description

This function formats the input string into a value representation, if possible. It formats the input string using the input scale factor, re-converts the value to a string, and then returns the formatted string value. If the input string cannot be converted, the input string is returned with no change to it.

Arguments

`t_string`	a string representing a float value
`t_scaleFactor`	a string representing a scale factor

Value Returned

nil	if the `t_scaleFactor` given is invalid
`t_val`	when the input string can be formatted using the input scale factor. Else, the input string is returned without any change to it.

Example

```
cdfFormatFloatString("123.4" "m") => "123400.0m"
cdfFormatFloatString("10000" "M") => "0.01M"
```

cdfSyncInstParamValue

```
cdfSyncInstParamValue(
     d_instId1
     d_instId2
     )
     => t / nil
```

Description

This function generates all the CDF parameters for the first instance (*instId1*) and updates the second instance (*instId2*) with the same values. Both the instances must share the same cell.

cdfUpdateInstSingleParam

```
cdfUpdateInstSingleParam(
     d_instId
     t_paramName
     )
     => t / nil
```

Description

This function copies the specified parameter's (*t_paramName*) effective value to the specified instance (*d_instId*).

Invoking the Edit CDF Form

You have the option of modifying how the Edit CDF form opens. You can use `aedEditCDF` to specify initial library, cell, and type values.

aedEditCDF

```
aedEditCDF(
     [?libName t_libraryName]
     [?cellName t_cellName]
     [?cdfType t_cdfType]
     )
     => t
```

Description

Opens the Edit CDF form to the library, cell, and CDF type specified by *`libraryName`*, *`cellName`*, and *`cdfType`*.
`libraryName` and *`cellName`* must be strings referring to an existing library or cell, and *`cdfType`* must be `'effective'`, `'base'`, or `'user'`.

cdfGetCustomViaCDF

```
cdfGetCustomViaCDF(d_customViaId)
     => cdfDataId / nil
```

Description

Returns the effective CDF description associated with a customVia or returns nil. When the customVia's cell or library has a base or user-level CDF defined, it returns the *`cdfDataId`*, otherwise returns `nil`.

cdfUpdateCustomViaParam

```
cdfUpdateCustomViaParam(d_customViaId)
     => t/ nil
```

Description

Stores the parameters specified in the effective cell CDF of the customVia in the specified customVia instance. When the specified ID is not for a customVia instance or the instance master does not have CDF definition, it returns `nil`.

9

Verifying CDF Operation

Testing the Component

Once you have completed the design of your component (all views and the Component Description Format are done), test the operation of the component's CDF description before using the component in your design. The following is a procedure you might use.

Test Procedure

1. Open an empty schematic and place instances of your symbol.
2. As you place the instances, use different parameter values.

   Check the Add Instance form for confirmation that your parameter definitions and callbacks are working. Look for the following indications:

   - ❑ Parameters have the units you expected.
   - ❑ Parameters have the defaults you expected.
   - ❑ The correct parameters are at the top of the list of parameters. You should not have to keep scrolling down the form to edit important parameters.
   - ❑ Parameters that are supposed to become noneditable do so when expected.
   - ❑ Parameters that are not supposed to be displayed are not displayed as expected.
   - ❑ Do your callbacks get called? First check. Do they stacktrace because you forgot to load them?
   - ❑ Parameters whose values depend on other parameters update as you expect.
   - ❑ Are your callbacks doing the range checking you set up?
   - ❑ Can you enter bad parameter values?
3. After you place the instances, use Edit Properties (in the schematic editor) to look at the actual properties stored on the instances.

   Look for the following:

   - ❑ Are you saving the default values?
   - ❑ Are nondisplayed parameters being set as you expected? If they are not displayed, it is hard to know that they worked.
4. Draw a minimal schematic and go to the simulator you will be using. Generate a netlist to make sure that your component netlists as you expect.

5. Once you are satisfied with the schematic and simulation results, go to an empty layout and repeat the CDF tests to see the effect on your layout.

Virtuoso XL Layout Editor Procedure

If you are going to use the Virtuoso® XL Layout Editor (Virtuoso XL),

1. Make a simple schematic with some examples of your component randomly wired to each other.
2. Go through the Virtuoso XL design cycle.
3. Check that you have set all the parameters that Virtuoso XL requires.
4. Does Virtuoso XL place the layouts you expect with appropriate connectivity?

LVS Procedure

If you are using Layout Versus Schematic (LVS),

1. Create a simple schematic and layout with very few instances.
2. Run extraction and LVS.
3. Make sure the LVS netlist is correct.

   This also gives you a chance to check out the LVS rules in a controlled experiment using just one device.

Making Changes

If you find a problem or change the design, follow these guidelines:

- If problems occur during testing, fix the views or CDF and retest.
- If the problem or change occurs after you have placed multiple instances of the component, consider the following:
  - You might need to change just the user interface, such as the order of the parameters or which parameters are displayed in the form or label display. When you make a change, it is inherited by all existing instances.
  - If you change the default value of a parameter with the attribute *storeDefault = no,* it is inherited by all existing instances. (Use the *Window – Redraw* command to see the changes reflected in the current window.)
  - If you change the default value of a parameter with *storeDefault = yes,* changing the default value does not update the existing instance, because the value was saved as a property on each instance.
  - If you change a callback that sets the value of one parameter based on an equation involving another parameter, changing the default value does not update the existing instances. You must also trigger the callback again, for each instance.

  In the last two cases, you can update existing instances using the *Cadence Analog Design Environment – Design Variables – Update* command. This command updates the default values and triggers all of the callbacks of all of the instances in the current window. The *Update* command has the following limitations:

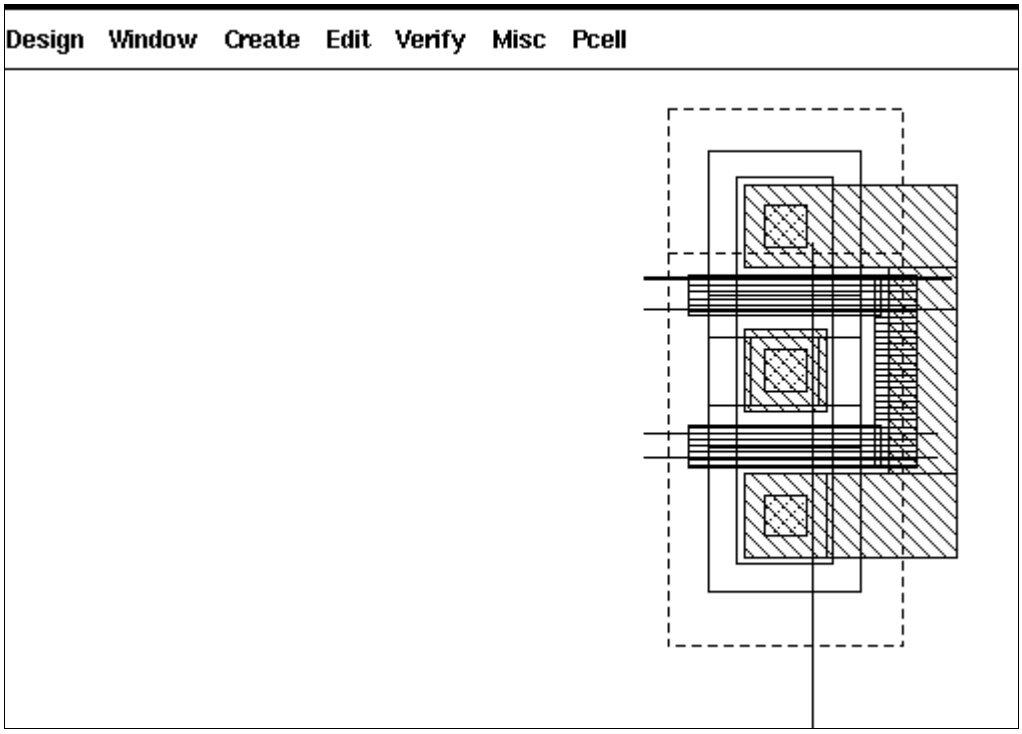
  - The update works only on the open cell.
  - The update does not traverse the design hierarchy.
  - There is no guaranteed order of running the callbacks, so you need to make sure that your callbacks work well together.

Finishing the NFET Example

In the last part of the NFET example you can see visible changes in the cell layout and form displays as a result of changing the CDF data.

Varying the Layout

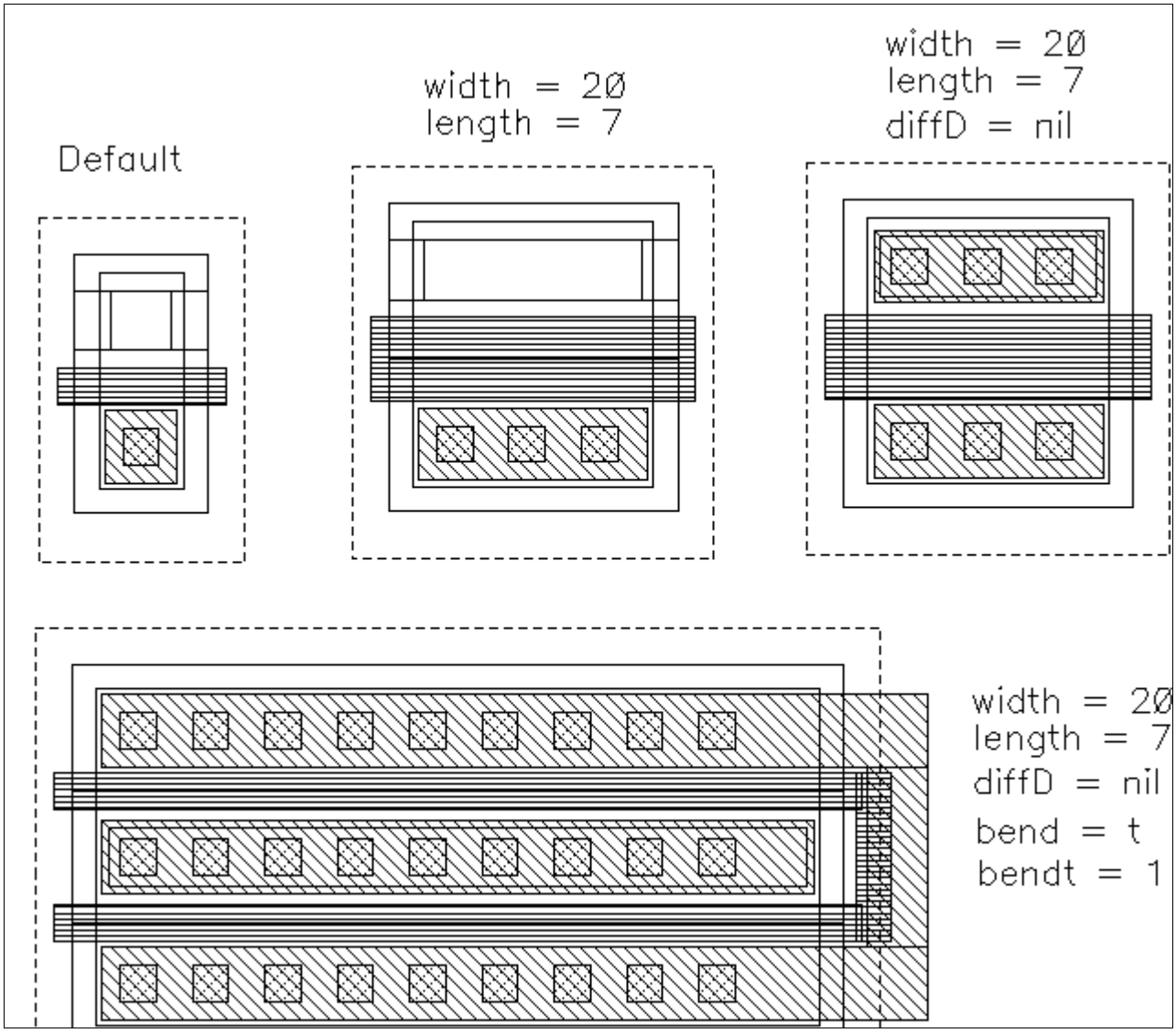
Once you enter all the CDF information for the NFET into the system, you can display the transistor layout. The *pcell* layout (stored in view *layout*) might look like the following:



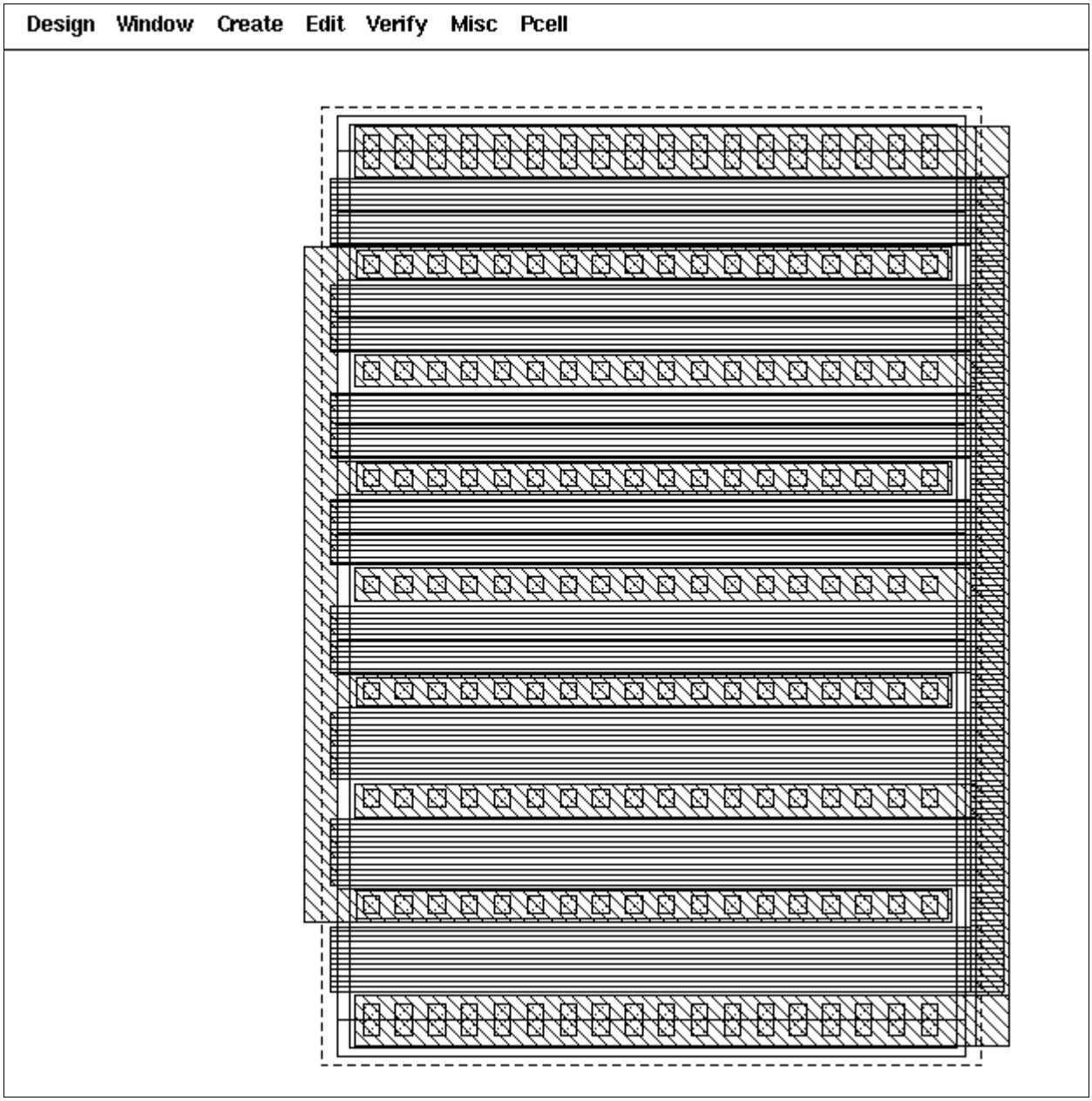
This is a *pcell* with the following parameters:

width	float, defining the width of the gates
length	float, defining the length of the gates
bend	Boolean, to control conditional inclusion of shapes for bend
bendt	float, to help in stretching shapes correctly when bend occurs
diffD	Boolean, to control conditional inclusion of shapes for diffusion drain contact versus metal drain contact

The NFET looks like the following with different parameter combinations:

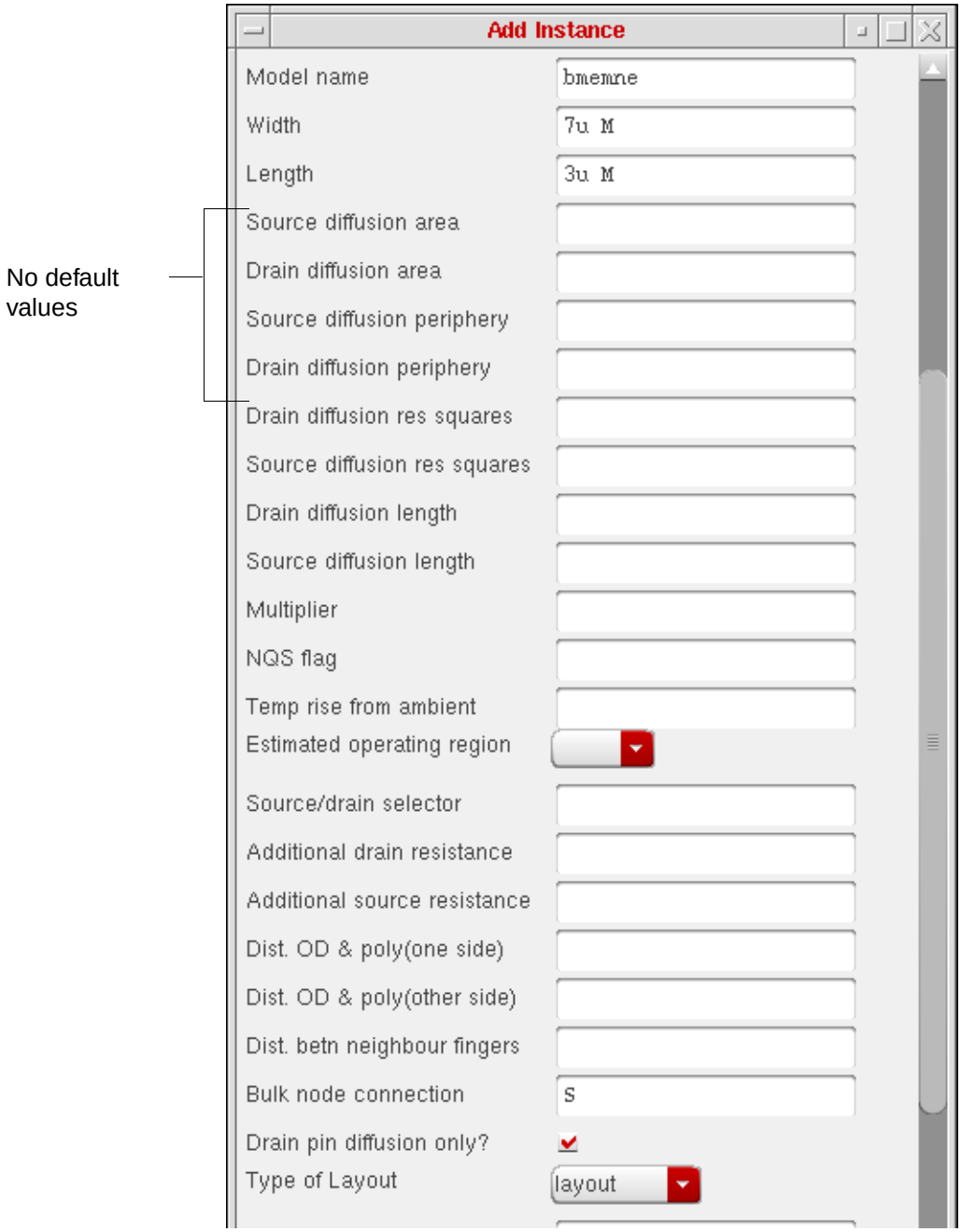


In addition, you need the fixed layout for the power applications. This might be stored under view *layout.pwr*, have a gate width of 880 micrometers and length of 12 micrometers, and look like this:



Changing Parameters

Now try to use the NFET in a design.



When you first use the Add Instance form, some fields are dimmed.

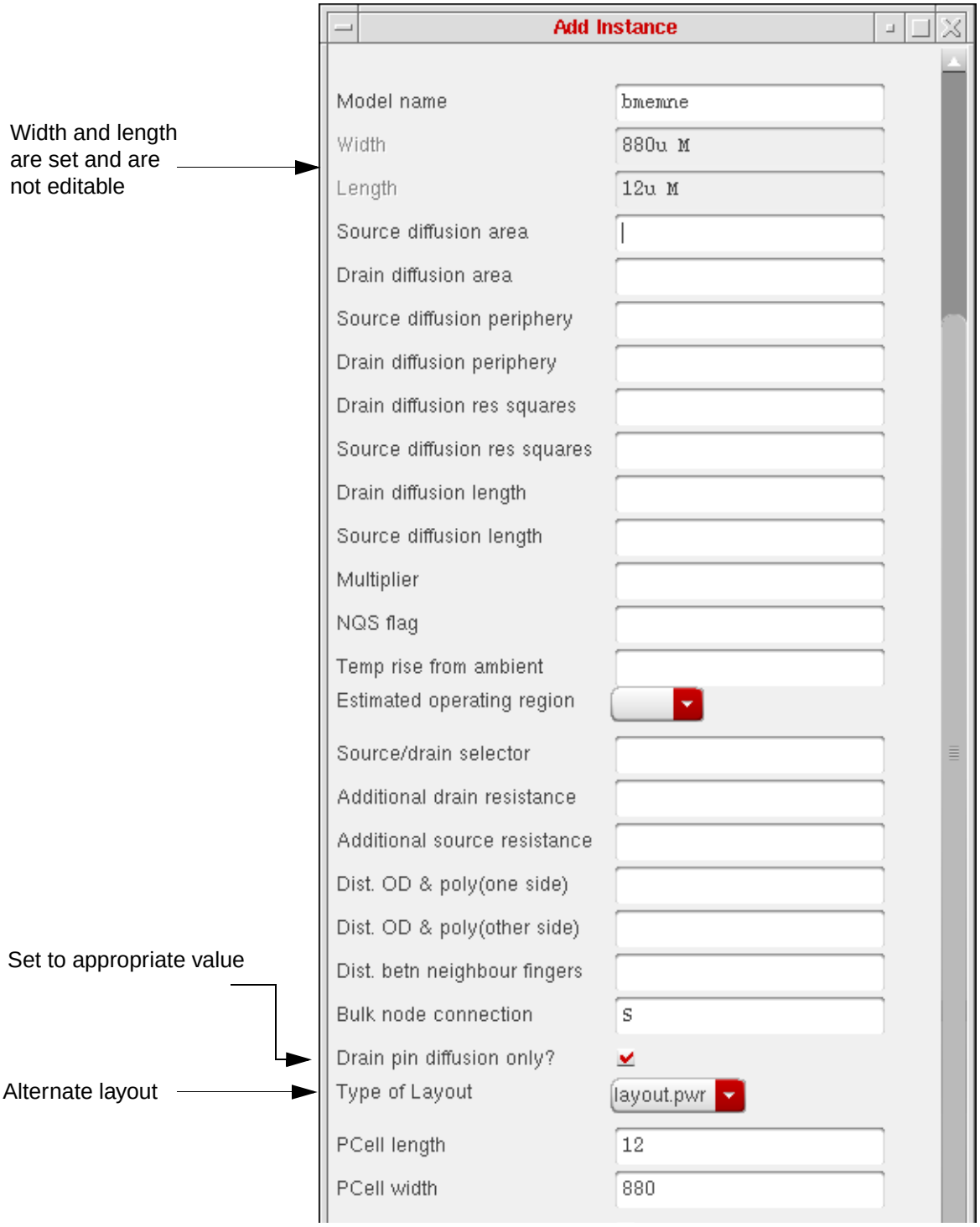
The first time you open the Add Instance form for your component, it shows all default values. Some default values are not defined. At this point, you might want to go back to the Edit CDF form and add the appropriate default values.

Notice, however, that you cannot enter values in the fields with dim names. If you use the CDF description to set a parameter to be displayed, but not editable, the parameter field is not editable on the Add Instance form.

The type of *layout* is *cyclic,* as you specified. The width and length are prompted for only once, as a string with units of length.

As you place instances of the NFET, you never change the value for the bulk node connection. Therefore, you might use the Edit CDF form to move that parameter down in the list of parameters until it is below other parameters that you do change.

1. Change the Type of Layout cyclic button. The Add Instance form updates as follows:



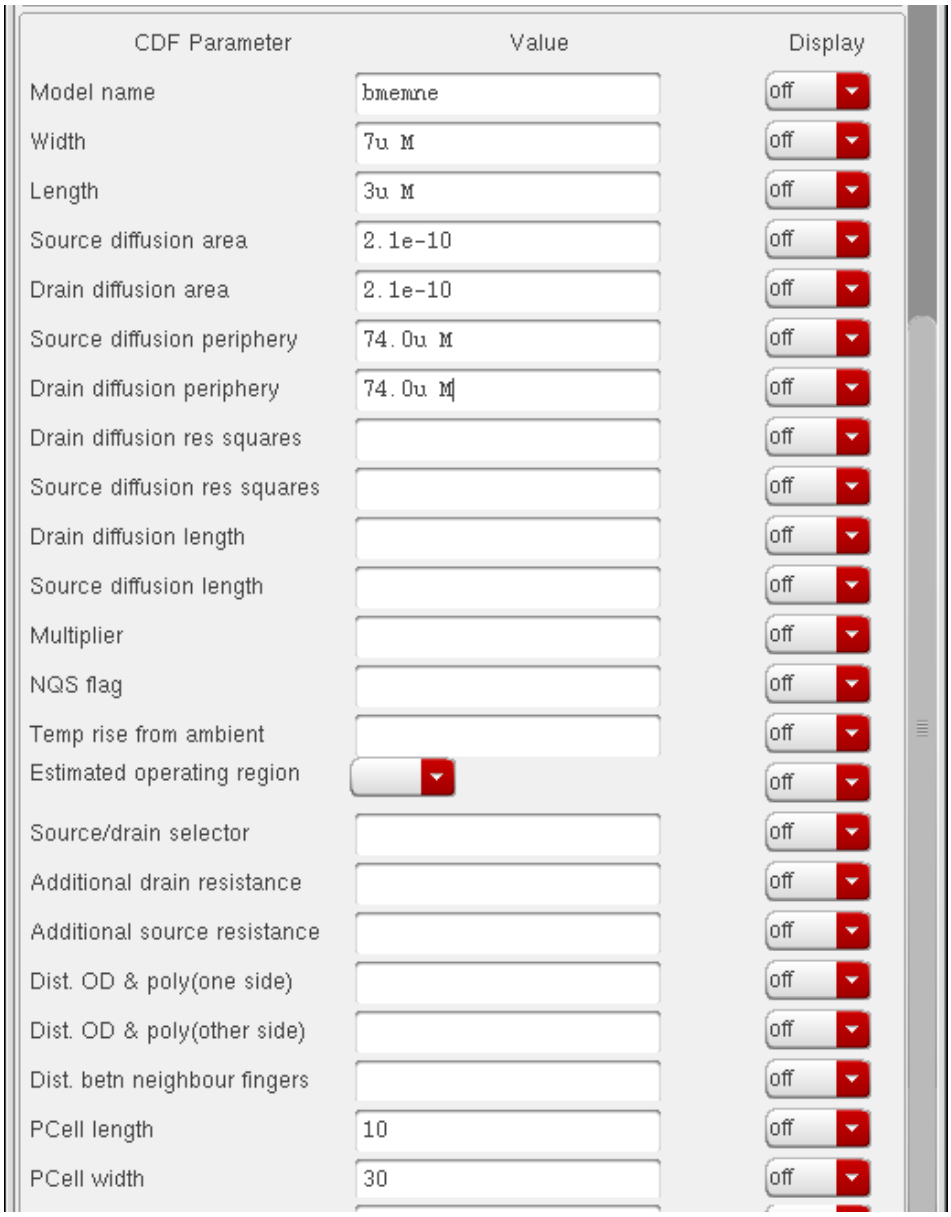
You can see the form modify itself to show the new fields, indicating that the callbacks are working.

At this point you still cannot see if the *pcell* parameters are being set properly.

2. Use the *Edit – Properties – Object* command on a few examples to verify the operation of the *pcell* parameters in the CDF description.

3. Using the Edit Object Properties form, create an instance with *width* set to `30u`, *length* set to `10u`, the default layout, and a drain diffusion pin.

   The CDF Parameter fields in the Edit Object Properties form should look like this:



CDF Parameter	Value	Display
Model name	bmemne	off
Width	7u M	off
Length	3u M	off
Source diffusion area	2.1e-10	off
Drain diffusion area	2.1e-10	off
Source diffusion periphery	74.0u M	off
Drain diffusion periphery	74.0u M	off
Drain diffusion res squares		off
Source diffusion res squares		off
Drain diffusion length		off
Source diffusion length		off
Multiplier		off
NQS flag		off
Temp rise from ambient		off
Estimated operating region		off
Source/drain selector		off
Additional drain resistance		off
Additional source resistance		off
Dist. OD & poly(one side)		off
Dist. OD & poly(other side)		off
Dist. betn neighbour fingers		off
PCell length	10	off
PCell width	30	off

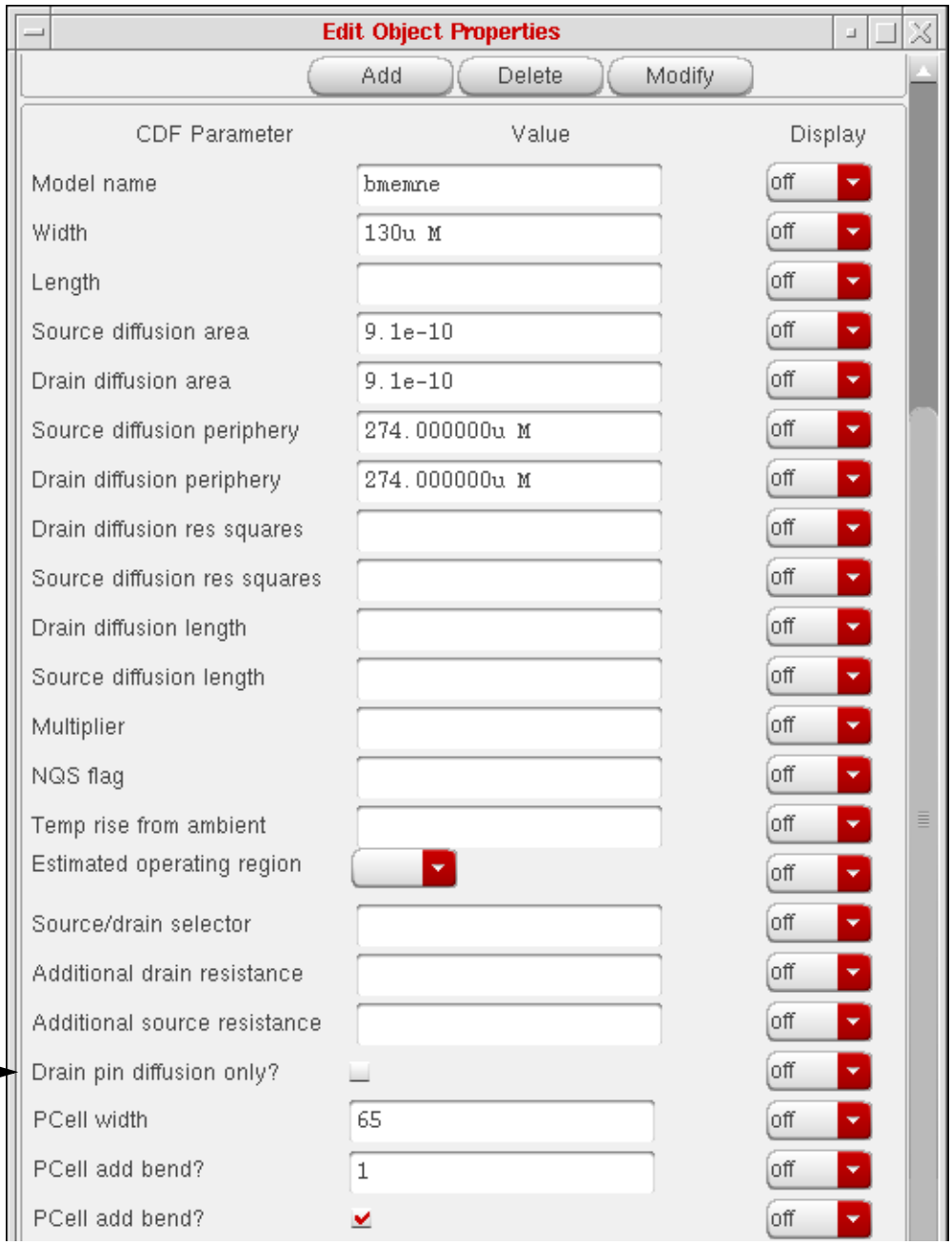
These values indicate that the callbacks used to generate *pcell* parameter values worked correctly. If you have a different result, check the callbacks in the CDF description.

4. Create an instance of the NFET with *width* set to `130u`, *length* set to `3u` (default), the default layout, and no drain diffusion pin (required).

The Edit Object Properties form should look like the following:

width is > 120, so bend is turned on

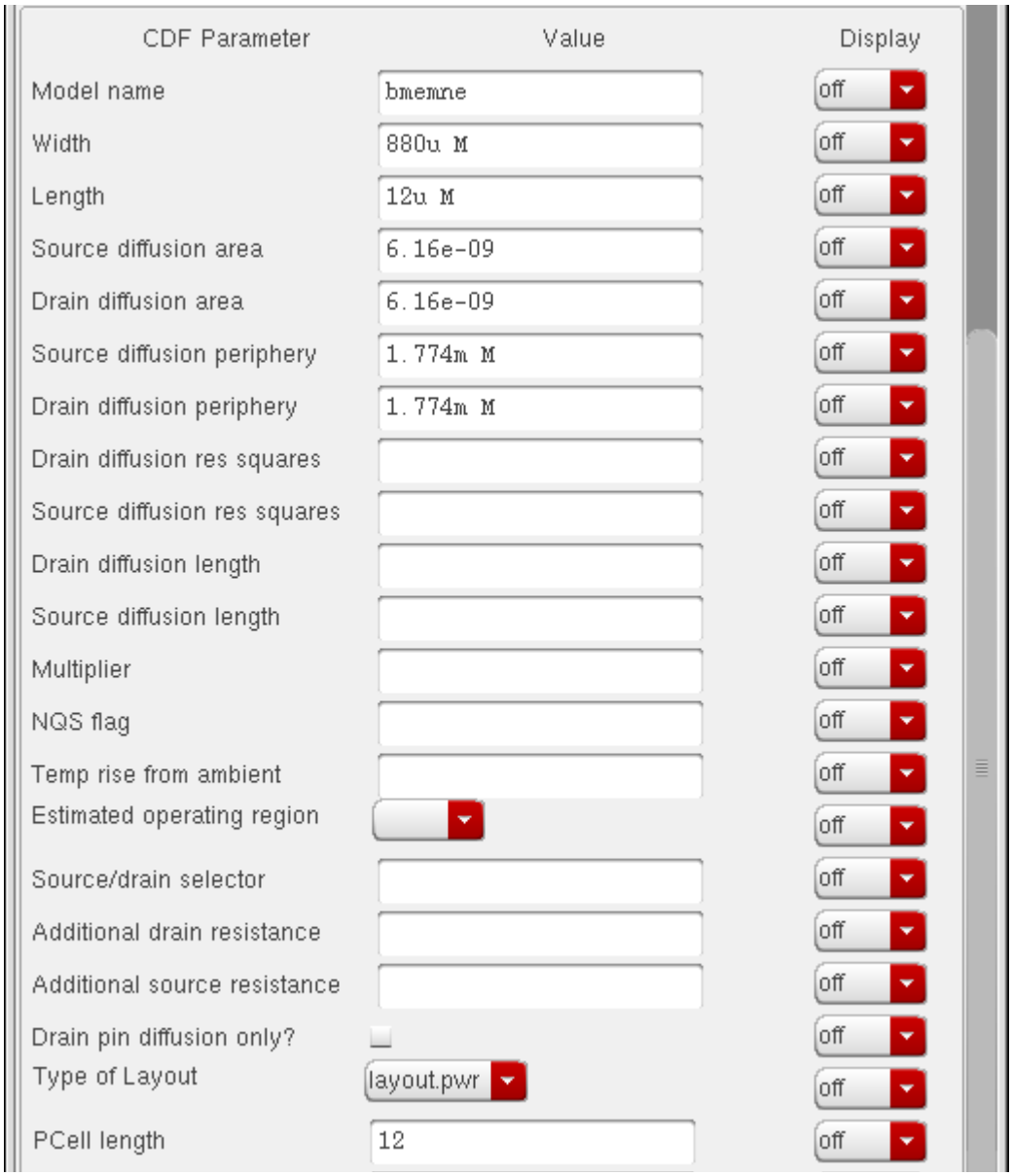
width is > 120, so no diffD



Both of the length parameters are default values, so no property is saved.

5. Create an instance with *maskLayoutViewName* as `layout.pwr`.

You see the following:



All the callbacks are triggered, as expected.

A

Advice and Warnings

This section answers frequently asked questions not already covered, and warns you about common mistakes.

- Library CDF versus Cell CDF

  A cell inherits all definitions from the library CDF. If there are changes in the cell CDF, the changes override the library CDF at the cell level. It is important to be aware of what is in the library CDF of any cell you work on.

- CDF level differences

  Remember that only the base-level CDF is saved to disk. The user-level CDF is used only for one session and cannot be saved. If you have a base and user CDF, the effective CDF is everything from the base CDF updated with anything new from the user CDF. In the Edit CDF form, you can “edit” the effective CDF. In reality, the effective CDF is just there to give you a way to look at the total CDF for a component. There is nothing to edit in an effective CDF, so changing the effective CDF in the Edit CDF form does nothing.

- Update

  The update function under *Cadence Analog Design Environment– Design Variables – Update* in the schematic editor updates only what is in the current window and for only one level of hierarchy.

- Parameter values and callbacks

  If you use *iPar*, *pPar, atPar, dotPar,* or the old notation ({}) to inherit the value of a parameter, as the real value changes, the callbacks associated with that parameter are NOT triggered. So, be aware of that when determining which parameters inherit values. Similarly, if you assign a design variable as a parameter value, the setting of the design variable does not trigger callbacks.

- Units

  The units you can use in the CDF are limited to those offered in the Edit CDF form. There are no user-defined units.

- Range checking

CDF provides no automatic range checking. You must use callbacks to do that.

- *Pcell* default values

  There is no system mechanism to ensure that your *pcell* defaults and CDF defaults are synchronized, so you must manage this.

- Callback files

  If you define your callbacks in a separate file (not self-contained in the CDF or attached to the library), you must remember to archive that file whenever you archive your library.

- Callback limitation

  You can control parameter values with callbacks, but you cannot affect anything in the simulation information section.

- Boolean properties

  Be careful of using the CDF “Boolean.” It is defined as *t* or *nil*. If you add a property to an instance using the *Add* button in the Edit Object Properties form you will see that the system models that kind of Boolean as TRUE or FALSE. There are some applications that depend on that. For example, the property *lvsIgnore* used in DLE and LVS must be TRUE or FALSE. So, if you want to add it to your CDF, instead of making it a CDF Boolean type, make it a cyclic type and set its choices to TRUE and FALSE.

- Triggering callbacks

  Callbacks are triggered whenever a change is made to the parameter value, even if the change is caused by loading a default value. You need to ensure that default values do not violate any relationships that you are trying to enforce.

  Use callbacks to establish dependencies between parameters whose values are numeric constants.

  Avoid using callbacks on parameters whose values are expressions. Such values require update evaluations and can quickly become out of date. If you do choose to use callbacks of this type, you will need to run a hierarchical evaluator that will walk the entire design to trigger all CDF callbacks whenever the value of the parameter is changed. Note, too, that a hierarchical evaluator will not be reliable with pPar() parameters.

- Layout extraction, properties, and callbacks

  When you run layout extraction to find devices in the layout, you can also measure the sizes of the devices. When you save that information, it is stored as a property on the symbol (*auLvs* or an *ivpcell*) that is placed in the extracted layout. For example, when extraction finds an NFET, it places the cell *nfet auLvs* at the location of the recognition shape. That instance has the connectivity that was found and has the properties for the

measured values saved, probably the length and width. Setting those measured parameters by extraction does not trigger any callbacks on those parameters. This is especially important to remember if you are doing parasitic resimulation, because you need all parameters necessary for simulation to be set in the extracted layout. You cannot rely on the callbacks; you must redo the calculations in the extraction rules.

- LVS case sensitivity

  Remember the property mapping example for LVS. Even though CDF might use lowercase parameter names, LVS expects capital letters. If you do not enter this mapping properly, the parameters are not netlisted correctly and in LVS this most likely means that size comparisons are not made at all.

- The Virtuoso® layout editor now accesses and uses CDF parameters.

- Property value replacement and callbacks

  Using *Edit – Search – Replace* to find a property and change its value across a set of instances does not trigger callbacks associated with the parameter. The *Cadence Analog Design Environment – Design Variables – Update* command triggers the callbacks associated with each variable, but you cannot control the order in which the callbacks are executed.

  In addition, if you use the default value for a parameter, and set *storeDefault* to *no*, there is no property for *Edit – Search – Replace* to find. However, changing the default value does change it.

- Numerical value format

  Be careful about entering blanks when entering values in a form or defining values or default values. In the CDF, “ 12” does not equal 12; nor does
  “12 ”. “12 u” does not equal “12u”. This can cause errors.

- The *dontSave* parameter attribute

  Do not use the *dontSave* parameter attribute unless you are absolutely sure you need to. It can lead to very strange results, especially when callbacks are used. The only thing that *dontSave* does for you is save the space used to store the property value on a parameter that you think is not very useful. One example is when you use the parameter *resType* on a resistor to trigger callbacks to set up the rest of the parameters based on the resistor type. Once that is done, you do not think the value of *resType* needs to be saved. The problems occur when you edit the parameters on existing instances and the values do not match.

- Reserved words

  There are words that are reserved and cannot be used as net, component, property, or parameter names. Do not use any of the following:

- Command or function names that your simulator uses
- A parameter or property name or prompt that is already in use
- The names of fields
- “others,” “loadSave,” “simInfo,” “othersFG,” “area,” “resistor,” or “multiplier”

In addition, if you want to make sure a property you define does not go into the netlist, do not use the name of a CDF parameter.

- Initial and final procedures

  Avoid using too many initial and final procedures (*initProc* and *doneProc*). Running one or two procedures every time you change a parameter can slow down system performance noticeably.

- Layout name labels

  Labels in layout do not know the hierarchy, so they cannot display hierarchical names, only current names.

- Do not use parameter names that begin with a number.
- Changes in the Edit CDF form do not take effect until you click *Apply* on the Edit CDF form.
- Avoid using the *dontUse* attribute.

B

Accessing Subcircuit Simulation Data

Overview

This appendix describes how to access simulation data from primitive subcircuits and schematics when you use the spectre interface. By modifying the Component Description Format (CDF) description of components in a design, you can annotate and plot subcircuit port currents, primitive model parameters, and operating point parameters. You can apply the techniques described here to any simulation primitive, not just subcircuits. These techniques allow you to annotate derived parameters on the schematic.

Text-based subcircuits are frequently used to

- Provide a better model of device behavior
- Allow quick and easy modification of the netlist representing the subcircuit

The first section describes how to modify a CDF description to obtain subcircuit port currents. The second section describes how to obtain the derived model parameters and operating point information. The third section describes how to obtain model parameters and operating point information from schematics below the level of the component in question. The last section reviews one method of editing a CDF description.

Note: Unless otherwise specified, the information in this appendix is for the *spectre* simulator. Other simulators may require a variation on the syntaxes detailed here.

The subcircuit examples use this simplified operational amplifier model. You can find this in the file

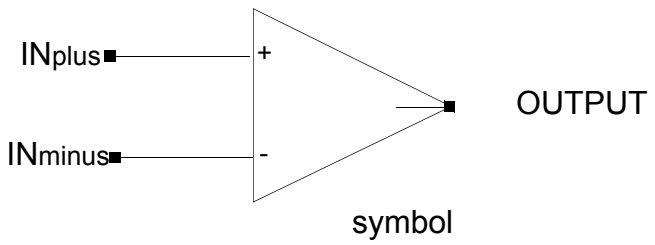
your_install_dir`/tools/dfII/samples/artist/models/spectre/vcvsOpampModels.scs`

```
subckt opamp (plus minus output)
parameters inputCap=500e-15
C1 (plus minus) capacitor c=inputCap
E1 (aout 0 plus minus) vcvs gain=1e6
R3 (aout output) r100m
R4 (plus minus) r10M
ends

model r100m resistor r=100e-3
model r10M resistor r=10e6
```

The subcircuit is represented at the next higher level in the design hierarchy by a symbol, with pins `INplus`, `INminus`, and `OUTPUT`. The instance `I16` in the cellview `vcvsOpampTest` schematic in the library `aExamples` references the cellview.

If you update the CDF, you must generate a new netlist in order for the enhancements described here to work.



Accessing Subcircuit Port Currents

You can obtain the port current of subcircuits of a component with simple simInfo on the CDF of that component. For the Spectre® circuit simulator, the use of expressions is not necessary because this simulator handles the calculation of currents.

Note: Only the port currents may be obtained. No access is provided to internal subcircuit currents.

The required simInfo is best illustrated by example. To provide access to the currents entering through the plus, minus, and output ports of the operational amplifier model for spectre, add the following Cadence® SKILL language lines to the text of the spectre Simulation Information section of the operational amplifier CDF description, using the symbol pin names `INplus`, `INminus`, and `OUTPUT`:

```
termMapping ( nil plus \:1 minus \:2 output \:3)
```

The *termMapping* property value is a disembodied property list (DPL*)*, which provides information to the data access code on how to obtain data for the relevant terminal. This information is also used by the simulator interface to generate appropriate save statements. The first term of the *termMapping* entry should always be *nil*. Note that spectre requires the order number of the terminal for subcircuits, not the name. So instead of `":plus"`, `":1"` is required.

The *termMapping* property describes that the schematic terminal `INplus` is mapped to plus, and `INminus` is mapped to minus, and `OUTPUT` is mapped to the simulator name output. The colon character (`:`) is the Spectre simulator's instance-terminal delimiter.

To save currents, use the Save Options form on the simulation window, or use the *Outputs – To Be Saved* command. You can select the individual terminals on the `I16` instance in the `vcvsOpampTest` schematic cellview. During simulation, only the currents you specify are saved.

Accessing Subcircuit Model Parameter and Operating Point Information

By modifying a device CDF description, you can save additional model parameters or operating point information using the following functions:

- **opParamExprList** for specifying DC operating point information
- **optParamExprList** for specifying transient operating point information
- **modelParamExprList** for specifying model parameters

After simulation, you can access this information from the calculator or by using the label display mechanism. This method differs from the one used in the previous example in that you must fully specify the required functions. This is demonstrated in the following example.

In this example, three additional operating point parameters are resistors: *rin* (input resistance), *rout* (output resistance), and *gain*. Add the following lines to the spectre Simulation Information section of the operational amplifier CDF description:

```
opParamExprList (
       ("rin"  "OP(mappedRoot(\".R4\") \"res\")")
       ("rout" "OP(mappedRoot(\".R3\") \"res\")")
       ("gain" "1e6")
)
```

The *opParamExprList* property is a list of pairs. The first element of each pair is the name of the new property, and the second element is the expression that must be evaluated to compute the new property. *OP* is the operating parameter function, which takes an instance name and an associated parameter as arguments. The second argument *res* is the resistance parameter for that component. The first argument, `mappedRoot(\".R4\")`, is the expression for the name of the instance.

The *mappedRoot()* function takes the string (`\".R4\"`) and prepends the simulator instance name of the current instance. The dot character (`.`) is the spectre subcircuit delimiter. Since the property value is a string, substrings (such as `".R4"`) must be escaped (`\".R4\"`).

The *root()* function prepends the schematic name (rather than simulator name) of the device instance in question. The *root()* function also adds a slash (the schematic delimiter) before its argument.

The other two functions provided, *inst()* and *mappedInst()*, do not take arguments and return the schematic and simulator instance names, respectively.

The following table shows the difference between the four functions.

Function Syntax	Value Returned
root(string)	strcat(instSchematicName "/" string)
mappedRoot(string)	strcat(instSimulatorName string)
inst()	instSchematicName
mappedInst()	instSimulatorName

The SKILL function *strcat* is the standard SKILL string concatenate function.

Given an instance of the subcircuit with the name `I16` and the simulator instance name of `I16`, the expressions in the example would evaluate as follows:

```
mappedRoot(\".R4\") -> "I16.R4"
root(\"inplus\") -> "I16/inplus"
```

The other functions would act as follows:

```
inst() -> "I16"
mappedInst() -> "I16"
```

Similarly, you can add transient operating point information and model parameters if you replace *opParamExprList* with *optParamExprList* and *modelParamExprList* respectively. Also replace *OP* with *OPT* and *MP* as follows:

```
OP(instance_name parameter)
OPT(instance_name parameter)
MP(instance_name parameter)
```

Following are examples of *optParamExprList* and *modelParamExprList*:

```
optParamExprList (("rin" "OPT(mappedRoot('.R4') 'res')"))
modelParamExprList (("afin" "MP(mappedRoot(\".R4\") \"af\")")
                    ("afout" "MP(mappedRoot(\".R3\") \"af\")"))
```

In the same way, you can add parameters to cells that are similar to the ones you find in *analogLib*. The following addition to a resistor CDF description adds a conductance property:

```
opParamExprList (
    ("cond"  "1.0/OP(inst() \"res\")")
)
```

You can define your own functions to perform computations. Consider the following example which adds the `xyz` parameter:

```
opParamExprList (
    ("xyz"  "userFun(inst())")
)
```

When the `xyz` operating point property is accessed, the function `userFun()` is called with the schematic name of the instance in question. The `userFunc()` should return an integer if the operation is successful. If `userFunc()` does not return an integer, then the parameter is not displayed. Some more examples of user defined functions are as follows.

```
opParamExprList (
    ("avgcurrent"   "averageCurrent(root('PLUS'))")
    ("pwr"          "instPwr(inst())")
)

defun( instPwr ( instName )
    (let (i v)
        i = OP(instName "i")
        v = OP(instName "v")
        (when (and i v)  i * v )
)

defun( averageCurrent (instTerm)
    (let (i)
        i = IT( instTerm)
        (when i average(i))
    )

)
```

While using a user a user defined function with `opParamExprList` ensure that the function is defined, otherwise it causes an error. This process can aid debugging.

The expression you use to define a given property can be arbitrary, but if the expression evaluates incorrectly or the current is not saved, the system generates an error. This can be slow down the simulation if there are many instances of the device on a schematic attempting to display this property. For example, you might add the following `avecurrent` property to a resistor to compute the average transient current through the resistor:

```
opParamExprList (
       ("avecurrent"  "average(IT(root(\"PLUS\")))")
)
```

If you did not save the current, then the `IT()` function returns `nil`, and the `average()` function complains about a `nil` argument. To prevent this, add logic to the expression as in this example:

```
opParamExprList (
       ("avecurrent"  "let(((i IT(root(\"PLUS\"))))
             (and i average(i)))")
)
```

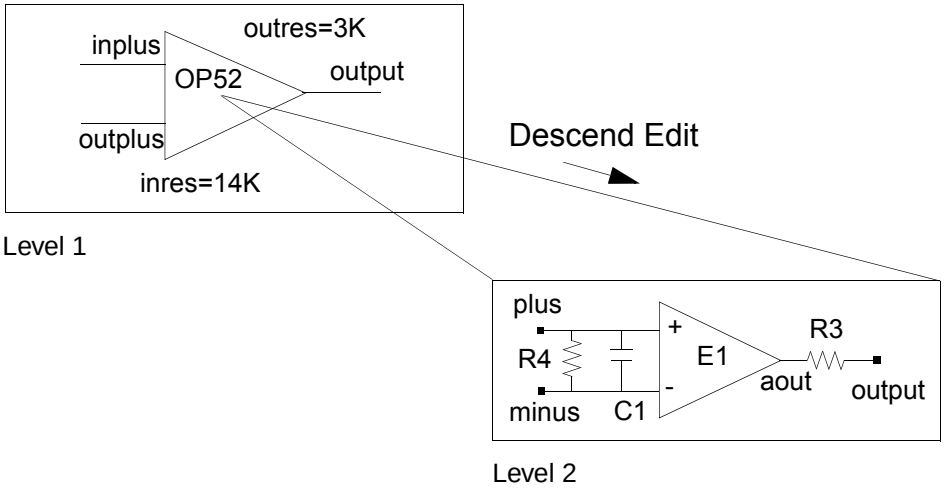
If you did not save the current, this expression prevents the system from flagging errors.

You can use standard SKILL functions like *let* in these expressions. *IT* and *average* are private, internal SKILL functions that are not documented for any other use.

Accessing Schematic Primitive Model and Operating Point Information

You can apply the methods for accessing model and operating point information in subcircuits to access that same information in schematics. By modifying the CDF description of a component, you can access model or operating point information in a schematic one level below the component and display it at the component's level.

For example, the instance `I19` in the cellview `vcvsOpampTest` is a schematic of the same subcircuit. On the next higher level is another schematic, where the sample schematic is represented by a symbol.



By putting a particular statement in the model or operating point expression list of the CDF *simInfo* section of *OP52*, you can define new parameters such as *inres* and *outres*, base them on primitive values one level down, such as *R4* and *R3*, and display their values on *OP52*.

To create the two parameters in the example, include these statements in the simInfo section of *OP52* for each simulator that you are using:

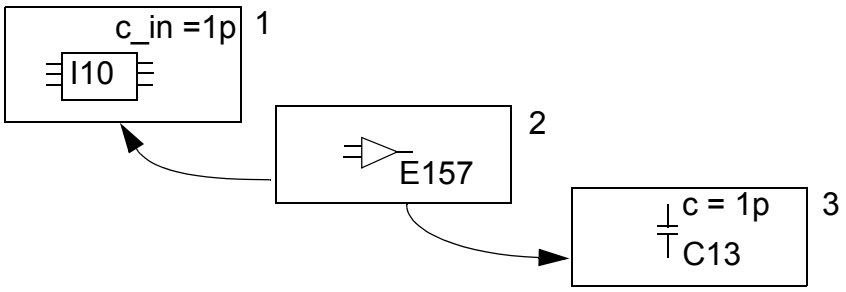
```
opParamExprList(
    ("inres" "OP(root(\"R4\") \"res\")")
    ("outres" "OP(root(\"R3\") \"res\")")
)
```

In this example, *inres* and *outres* are parameter names that you define. They derive their values from the *res* parameter on primitives *R4* and *R3*, respectively. Note that for schematics you use the *root* function.

To display the information produced by these CDF description entries, make sure that you have created enough labels for your component and that those labels are set to display model or operating point values. For model data, use *modelParamExprList* and the *MP* function.

Note: CDF data is not view specific, so do not reference parameters in views that are not netlisted because of their position in the view switch list.

To refer to primitive data several design levels below the component where you want the data displayed, you need to include all the instance names from each level.



For this example, use the following statement in the CDF description of `I10` to display the capacitance on `C13` as the input capacitance of `I10`.

```
opParamExprList (
    ("cin" "OP(root(\"E157/C13\") \"c\"))
)
```

Editing a CDF Description

Writable Cells

To modify a cell CDF description for which you have write access

1. In the CIW, type

   `cdfDump( "lib" "`*`full_path/filename`*`" ?cellName "cell" ?edit 't ?level 'base)`

   and wait for a window to appear.

2. Edit the file, making the necessary changes.

3. Load the file in the CIW with

   `load("`*`full_path/filename`*`")`

Sample CDF

The `aExamples` library demonstrates several different methods of parameter display.

The following is a sample CDF for several of the schematics in the `aExamples` library. Spectre examples are given:

- opamp (macro model)

```
cdfId->simInfo->spectre = '( nil
        propMapping       nil
        instParameters    nil
        otherParameters   (model)
        netlistProcedure  nil
        termMapping      ( nil IN\plus ":1" IN\minus ":2" OUTPUT ":3")
        termOrder         ("INplus" "INminus" "OUTPUT")
        componentName     opamp
        opParamExprList    (("rin" "OP(mappedRoot(".R4") "res")") ("rout"
"OP(mappedRoot(".R3") "res")") ("gain" "1e6"))
    )
```

- opamp_sch (1 level display):

```
cdfId->simInfo->spectre = ’( nil
        termOrder         ("INplus" "INminus" "OUTPUT")
        componentName nil
        opParamExprList   (("inres" "OP(root("R2") "res")") ("outres"
"OP(root("R3") "res")"))
    )
```

- opamp_2Level (2 level display):

```
cdfId->simInfo->spectre = ’( nil
        termOrder         ("in_plus" "in_minus" "out_again")
        componentName     "subcircuit"
        opParamExprList   (("cin" "OP(root("I0/C0") "cap")") ("rin"
"OP(root("I0/R2") "res")"))
    )
```

C

NBSIM Transistor CDF SKILL Description

Dumping the CDF description of the *nbsim* cell in the *analogLib* library produces the following:

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

let( ( libId cellId cdfId )

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

    cdfId = cdfCreateBaseCellCDF( cellId )

;;; Parameters
cdfCreateParam( cdfId
    ?name "model"
    ?prompt "Model name"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('model)"
    ?parseAsCEL "yes"
)

 cdfCreateParam( cdfId
     ?name "bn"
     ?prompt "Bulk node connection"
     ?defValue "S"
     ?type "string"
     ?display "artParameterInToolDisplay('bn)"
     ?parseAsCEL "yes"
 )

 cdfCreateParam( cdfId
     ?name "m"
     ?prompt "Multiplier"
     ?defValue "1"
     ?type "string"
     ?display "artParameterInToolDisplay('m)"
     ?parseAsNumber "yes"
     ?parseAsCEL "yes"
 )

 cdfCreateParam( cdfId
     ?name "w"
     ?prompt "Width"
```

```
    ?units "lengthMetric"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('w)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "l"
    ?prompt "Length"
    ?units "lengthMetric"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('l)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "ad"
    ?prompt "Drain diffusion area"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('ad)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "as"
    ?prompt "Source diffusion area"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('as)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "pd"
    ?prompt "Drain diffusion periphery"
    ?units "lengthMetric"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('pd)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "ps"
    ?prompt "Source diffusion periphery"
    ?units "lengthMetric"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('ps)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "nrd"
    ?prompt "Drain diffusion res squares"
```

```
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('nrd)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "nrs"
    ?prompt "Source diffusion res squares"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('nrs)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "ld"
    ?prompt "Drain diffusion length"
    ?units "lengthMetric"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('ld)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "ls"
    ?prompt "Source diffusion length"
    ?units "lengthMetric"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('ls)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "off"
    ?prompt "Device initially off"
    ?type "boolean"
    ?display "artParameterInToolDisplay('off)"
)
cdfCreateParam( cdfId
    ?name "Vds"
    ?prompt "Drain source initial voltage"
    ?units "voltage"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('Vds)"
    ?parseAsNumber "yes"
    ?parseAsCEL "yes"
)
cdfCreateParam( cdfId
    ?name "Vgs"
    ?prompt "Gate source initial voltage"
    ?units "voltage"
    ?defValue ""
    ?type "string"
    ?display "artParameterInToolDisplay('Vgs)"
```

```
     ?parseAsNumber "yes"
     ?parseAsCEL "yes"
 )
 cdfCreateParam( cdfId
     ?name "Vbs"
     ?prompt "Bulk source initial voltage"
     ?units "voltage"
     ?defValue ""
     ?type "string"
     ?display "artParameterInToolDisplay('Vbs)"
     ?parseAsNumber "yes"
     ?parseAsCEL "yes"
 )
 cdfCreateParam( cdfId
     ?name "trise"
     ?prompt "Temp rise from ambient"
     ?defValue ""
     ?type "string"
     ?display "artParameterInToolDisplay('trise)"
     ?parseAsNumber "yes"
     ?parseAsCEL "yes"
 )
 cdfCreateParam( cdfId
     ?name "region"
     ?prompt "Estimated operating region"
     ?defValue "triode"
     ?choices '("off" "triode" "sat" "subth")
     ?type "cyclic"
     ?display "artParameterInToolDisplay('region)"
     ?parseAsCEL "yes"
 )
 cdfCreateParam( cdfId
     ?name "rdc"
     ?prompt "Additional drain resistance"
     ?units "current"
     ?defValue ""
     ?type "string"
     ?display "artParameterInToolDisplay('rdc)"
     ?parseAsNumber "yes"
     ?parseAsCEL "yes"
 )
cdfCreateParam( cdfId
     ?name "rsc"
     ?prompt "Additional source resistance"
     ?units "current"
     ?defValue ""
     ?type "string"
     ?display "artParameterInToolDisplay('rsc)"
     ?parseAsNumber "yes"
     ?parseAsCEL "yes"
 )
cdfCreateParam( cdfId
     ?name "dtemp"
     ?prompt "Temperature difference"
     ?defValue ""
     ?type "string"
     ?display "artParameterInToolDisplay('dtemp)"
     ?parseAsNumber "yes"
```

```
     ?parseAsCEL "yes"
 )
cdfCreateParam( cdfId
     ?name "geo"
     ?prompt "Source/drain selector"
     ?defValue ""
     ?type "string"
     ?display "artParameterInToolDisplay('geo)"
     ?parseAsNumber "yes"
     ?parseAsCEL "yes"
 )
;;; Simulator Information
cdfId->simInfo = list( nil )
cdfId->simInfo->auCdl = '( nil
     netlistProcedure        ansCdlCompPrim
     instParameters          (m L W)
     componentName           nbsim
     termOrder               (D G S progn(bn))
     propMapping             (nil L l W w)
     namePrefix              "M"
     modelName               "NM"
)
cdfId->simInfo->auLvs = '( nil
propMapping              nil
netlistProcedure         ansLvsCompPrim
instParameters           (m l w)
componentName            nbsim
termOrder                (D G S progn(bn))
deviceTerminals          "D G S B"
permuteRule              "(p D S)"
namePrefix               "Q"
)
cdfId->simInfo->cdsSpice = '( nil
netlistProcedure         ansSpiceDevPrim
instParameters           (m w l ad as pd ps nrd nrs ld ls off vds vgs vbs)
otherParameters          (model bn)
componentName            bsim
termOrder                (D G S progn(bn))
propMapping              (nil vds Vds vgs Vgs vbs Vbs)
namePrefix               "S"
current                  port
dcSens                   t
acSens                   t
)
cdfId->simInfo->hspiceS = '( nil
netlistProcedure         ansSpiceDevPrim
instParameters           (l w ad as pd ps nrd nrs rdc rsc off Vds Vgs Vbs dtemp
    geo m)
otherParameters          (bn model)
componentName            hnmos
termOrder                (D G S progn(bn))
propMapping              (nil vds Vds vgs Vgs vbs Vbs)
namePrefix               "M"
current                  port
dcSens                   t
acSens                   t
)
```

```
cdfId->simInfo->spectre = '( nil
propMapping              nil
otherParameters          (bn model)
instParameters           (w l as ad ps pd nrd nrs ld ls m trise
    region)
termOrder                (D G S progn(bn))
termMapping              (nil D d G g S s B b)
)

cdfId->simInfo->spectreS = '( nil
propMapping              nil
netlistProcedure         ansSpectreSDevPrim
otherParameters          (bn model region)
instParameters           (w l as ad ps pd nrd nrs ld ls m trise)
termOrder                (D G S progn(bn))
termMapping              (nil D d G g S s B b)
namePrefix               "S"
componentName            bsim1
current                  port
)

;;; Properties
cdfId->formInitProc              = ""
cdfId->doneProc                  = ""
cdfId->buttonFieldWidth          = 340
cdfId->fieldHeight               = 35
cdfId->fieldWidth                = 350
cdfId->promptWidth               = 175
cdfId->modelLabelSet             = "vfb phi eta"
cdfId->opPointLabelSet           = "id vgs vds"
cdfId->paramLabelSet             = "-model l w"
cdfSaveCDF( cdfId )

)
```

Glossary

A

AEL

Analog Expression Language. A syntax for expressions that allows the rapid evaluation of those expressions. AEL lets you express values as mathematical expressions instead of single, fixed values. Refer to the *Analog Expression Language Reference*.

auCdl

Analog and Microwave CDL. The analog and microwave version of CDL, which originally only ran on digital designs.

auLvs

Analog and Microwave LVS. The analog and microwave version of LVS, which originally only ran on digital designs.

B

base level

The fundamental representation of a cell or library CDF description. Base level and user level are combined to produce the effective level, with user-level values overriding base-level values. Only the base level is permanently stored to the system disk.

C

callback procedure

A procedure triggered by a change in a parameter in a CDF description. Callbacks can be SKILL procedures, expressions, or functions entered directly into the Edit Component Parameter form, or they can be calls to procedures located elsewhere.

CDF

Component Description Format. A system for dynamically changing and storing parameters and displaying information for components and sets of components for different versions (levels) of designs.

CDL

Circuit Description Language. CDL is a subset of SPICE used by Dracula®, with extensions that allow for such things as global signals and empty cells.

CEL

CDF Expression Language. Another name for AEL.

cell

The Cadence software representation of a design element (or component).

CIW

Command Interpreter Window. The primary user interface for launching Cadence software. The Edit CDF form starts from the Tools menu in the CIW. You can enter SKILL commands in the CIW command line.

component

A fundamental unit within a system that encapsulates behavior and/or structure (also known as an *element*). A cell, with cellviews and associated CDF.

cyclic field

A type of form field that displays only one value, but reveals a sliding menu when you select the field.

D

disembodied property list

A software record used to store and retrieve component properties. New properties can be dynamically added or removed. A disembodied property list starts with a SKILL data object, usually *nil*, followed by alternating name and value pairs.

E

effective level

A masked sum of all the CDF values for a cell or library.

F

form field

The empty rectangle on a form where you type in values and names.

H

hpmns

MNS™ microwave design simulator from Hewlett-Packard.

I

interpreted labels

Displayed component values which change with each instance or change as the component parameters change.

instance parameters

Those parameters used in a module description that change with each instantiation of that module. These parameters are defined in the first section of a module, the module interface declarations, and are specified each time a module is called in a netlist instance statement.

L

libra

Libra®. A microwave design simulator from EESof.

LVS

Layout Versus Schematic. A Dracula verification tool that confirms that your layout matches your original schematic.

M

mharm

Microwave Harmonica®. A microwave design simulator from Compact Software.

model

A named instance with a unique group of parameters specifying the behavior of one particular version of a component. You can use models to instantiate components with parametric specifications different than those in the original component definition.

model parameter

Those parameters in a module description that remain the same with each instantiation of the original module, but which are designed to be changed when the module is called by a model. These parameters are defined in the second section of a module, the global

module scope declarations, and they are specified each time a module is called in a model instance statement.

N

nil

No value (an absence of value) or a *false* value.

P

parameter

A characteristic of a component.

parameter attributes

A characteristic of a parameter.

parse

The system evaluation of CDF information. Values can be parsed in absolute terms or according to a predefined system.

pcell

A parameterized cell. You can change a set of cells by changing one parameter.

prompt

The title or words to the left of the form field or button that tell you the name or nature of the parameter or other value that you enter at that place in the form.

property

A piece of information about an instance or object in a design.

R

radio button

An entry on a form that behaves as a push button does. You click on the button to turn the function on (the symbol turns black) and off (the symbol turns white).

radio field

A set of interlinked radio buttons in a single form field associated with one parameter. Typically, only one button can be on at any one time.

S

SKILL

A proprietary Cadence programming language based on over 1,200 functions. Each function follows a standard SKILL syntax and acts on a particular data type or data object.

SKILL command

When SKILL functions or expressions are used alone or entered into the CIW command line, they are often called SKILL commands.

SKILL expression

An equation where a data object is set equal to a SKILL function. AEL expressions can be used to express individual values in a SKILL function.

SKILL function

The fundamental element in the SKILL language. An operation to be performed (usually specific to Cadence software), usually followed by the items or data objects that the operation is to be performed on.

SKILL procedure

A short program or “script” or routine made up of multiple SKILL expressions and functions.

SKILL routine

A program made up of multiple SKILL expressions and functions.

T

A non-*nil* or *true* value. Used in Boolean values.

U

user level

A temporary level of cell or library CDF information that overrides base-level information, but is lost at the end of each design session unless you take specific actions to save it.

V

VLE

Virtuoso Layout Editor. Cadence's integrated circuit layout editor that supports IC cell layout design down to the polygon level.

W

watscad

WATSCAD™. A switched capacitor design simulator from the University of Waterloo.

Index

Symbols

A

B

C

D

E

F

T

U

V

W