Advanced Design System 2002

Design Kit Model Verification

February 2002

Notice

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

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

Warranty

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

Restricted Rights Legend

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

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

Copyright © 2002, Agilent Technologies. All Rights Reserved.

Acknowledgments

MS-DOS®, Windows®, and Windows NT® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of the Open Group.

Cadence® and Spectre® are registered trademarks of Cadence Design Systems Inc. Copyright © 2001 Cadence Design Systems Inc. All rights reserved.

Copyright © 2001 Avant! Corporation. All rights reserved.

Copyright © 2001 Red Hat, Inc. All rights reserved.

Compilation Copyright © 1998-2001, O'Reilly & Associates, Inc. All Rights Reserved.

Contents

1 Introduction

Advanced Design System ... 1-1
ADS Design Kit Model Verification Tool ... 1-1
Intended Audience ... 1-2
The Design Kit Model Verification Tool Process ... 1-2
What's in this Manual ... 1-4
Minimum ADS Installation Requirements ... 1-4

2 Setting up the Design Kit Model Verification Tool

Supported Simulators ... 2-1
Using Perl ... 2-1
- Using Perl on UNIX ... 2-3
- Using Perl on Windows ... 2-3

Configuring the Environment ... 2-4
Prior to Running a Script ... 2-5

3 Building a Basic Script

Understanding Items and Handles ... 3-1
Developing the Script Header ... 3-2
Understanding Templates ... 3-4
- Installing the Default Templates ... 3-4
- Creating Custom Templates ... 3-5

Understanding Parameters ... 3-8
- Installing the Default Parameters ... 3-8
- Creating Custom Parameters ... 3-8

Creating a Test Circuit ... 3-10
- Setting up a Test Circuit ... 3-11
- Adding Items ... 3-12
- Adding Circuit Options ... 3-12
- Adding Model Library Files ... 3-13
- Creating Instances ... 3-14
- Connecting Instances ... 3-15
- Creating Analyses ... 3-15
- Defining the Simulators ... 3-18
- Running the Test Circuit Script ... 3-22

Viewing Your Results in a Data Display ... 3-23

4 Building an Advanced Script

Module Reference ... 4-1
Using the Module Debug Facility ... 4-1
Quoting your string-variables ... 4-2
Template Module ... 4-3
- Creating a New Template Handle ... 4-4
- Retrieving a Handle of an Existing Template ... 4-4
- Documenting a Template ... 4-4
- Retrieving the Template Documentation ... 4-4
- Setting the Item Name Prefix ... 4-4
- Defining the Translation Rules ... 4-5
- Defining Template Parameters and Nodes/Pins ... 4-7
- Setting a Default Template Parameter Value ... 4-7
- Retrieving the Default Template Parameter Value ... 4-7
- Creating a Template for a Subcircuit ... 4-7
- Displaying the Defined Templates ... 4-8
- Saving the Template Definitions to Allow Fast Access ... 4-8

Parameter Module ... 4-9
- Creating a Parameter Handle ... 4-11
- Documenting a Parameter ... 4-11
- Retrieving the Parameter Documentation ... 4-11
- Defining the Dialect Name for a Parameter ... 4-11
- Defining the Result Name for a Parameter ... 4-12
- Displaying the Defined Parameters ... 4-12
- Saving the Parameter Definitions to Allow Fast Access ... 4-13
- List All Missing Parameter Definitions ... 4-13

Circuit Module ... 4-14
- Creating a New Circuit Handle ... 4-15
- Setting the Circuit Name ... 4-15
- Adding Simulator Options ... 4-15
- Removing Simulator Options ... 4-15
- Adding a Path to a Model Library ... 4-16
- Removing a Path to a Model Library ... 4-16
- Adding Components/Instances ... 4-16
- Removing Components/Instances ... 4-17
- Adding Analyses ... 4-17
- Removing Analyses ... 4-17
- Simulating a Circuit ... 4-17
- Adding a Startdeck and Enddeck Card ... 4-18
- Defining and Modifying SubCircuit Parameters ... 4-18
- Creating the Simulator Netlist ... 4-18
- Simulating a Circuit from a Netlist ... 4-19

Freeing Memory Occupied by the Simulation Results 4-19
Instance and Analysis Modules 4-21
Creating a New Item Handle 4-22
Viewing the Template Definition 4-22
Defining and Retrieving the Parameter Values 4-22
Retrieving the InstanceName 4-23
Defining and Retrieving Node Names 4-23
Retrieving the AnalysisName 4-23
Adding a Sweepplan 4-23
Removing a Sweepplan 4-24
Adding an OutputPlan 4-24
Removing an OutputPlan 4-24
Adding a Sub Analysis 4-24
Removing a Sub Analysis 4-25
Scale/Offset a Sweep Parameter 4-25
Results Module 4-27
Set/Get the Offset of a Sweepvariable when Processing the Simulation Results 4-28
Set/Get the Scale of a Sweepvariable when Processing the Simulation Results 4-28
Reading the Simulation Results or Citifiles 4-29
Retrieving a Handle to the Results 4-29
Writing the Results to Citifiles 4-29
Converting Citifiles to Datasets 4-30
Comparing Two Simulator Results 4-30
Performing a Simple Statistical Analysis 4-31
Merging Citifiles into One Big Citifile 4-31
Freeing the Memory taken by the Simulation Results 4-31
Design Kit Model Verification Tool Perl Modules 4-32
Template Module (dKitTemplate.pm) 4-33
Parameter Module (dKitParameter.pm) 4-35
Circuit Module (dKitCircuit.pm) 4-36
Instance Module (dKitInstance.pm) 4-38
Analysis Module (dKitAnalysis.pm) 4-39
Results Module (dKitResults.pm) 4-41
Example Device Test Scripts 4-43
diode_test_dc.pl 4-43
diode_create.pl 4-46
diode_parameter.pl 4-48

5 Comparing Simulation Results

Viewing Your Results in a Data Display 5-1
Script Based Comparisons 5-1
Example Script 5-2

6 Documenting Your Simulation Results

Creating Verification Documentation 6-1
Adding a Page 6-3
Notebook Operations 6-4
Page Operations 6-5
Page Creation and Deletion 6-5
Notebook Generation 6-5

A Template List

Analysis Templates A-2
AC analysis A-2
DC analysis A-2
Noise analysis A-3
S-parameter analysis A-3
SWEEP analysis A-3
Outputplan Templates A-4
Outputplan for Device Currents A-4
Outputplan for Device Operating Point (DOP) A-4
Outputplan for Nodes A-4
Sweepplan Templates A-6
Linear Sweep Plan A-6
Discrete Point Sweepplan A-7
Simulator Options Templates A-8
Circuit Simulator Options A-8
Circuit Temperature A-8
ModelLibrary Template A-9
Model Library A-9
Components/Instance Templates A-10
Parameter A-10
Capacitor A-10
Capacitor with Model A-11
Diode A-12
Independent Current Source A-13
JFET A-14
Mutual Inductor A-15
Inductor A-16
Inductor with Model A-17
MOSFET A-18
PORT A-19
BJT A-20
Resistor with Model A-21
Independent Voltage Source A-22

MESFET A-23
Circuit Initialization.. A-24
Circuit End Deck Card A-24
Command to Invoke Simulator A-24
Netlistnames for the Different Simulators A-25
Circuit Start Deck Card... A-25
Index

Chapter 1: Introduction

Many designers would like to take advantage of the powerful simulation capabilities of Advanced Design System (ADS) from Agilent Technologies. In order to use ADS effectively, designers must have access to a set of required models. Agilent Technologies is addressing this need by providing the translated ADS model files for distribution in the form of ADS design kits. With the information provided in this document, the models used in these ADS design kits can be verified against other simulator models such as Hspice and Spectre.

Advanced Design System

Advanced Design System has been developed specifically to simulate the entire communications signal path. This unique solution integrates the widest variety of proven RF, DSP, and electromagnetic design tools into a single, flexible environment. Building on years of expertise developing new technologies for our EDA tools, such as Series IV and MDS, Advanced Design System provides a broad range of high-performance capability. This makes it easy to explore design ideas, then model the electrical and physical design of the best candidates.

ADS Design Kit Model Verification Tool

Verification of the translated design kit models is one of the most important steps in the overall design kit creation process. This step is typically the most technically challenging and time consuming. The ADS Design Kit Model Verification process utilizes a library verification tool that helps to simplify many of the steps required to verify device models for use with a foundry's device and process parameter sets. This Perl script driven tool is used to verify that results from the ADS simulator match the output from the native simulator for design kit models that were translated from other formats such as Hspice or Spectre.

The model verification tool verifies that the outputs from the ADS simulator match the results from the foundry benchmark data to within a specified margin of error. The foundry benchmark data is typically the output from SPICE simulations of the foundry's test circuits. For more information on simulators supported by the Design Kit Model Verification tool, refer to "Supported Simulators" on page 2-1.

Intended Audience

The audience intended for this manual consists of people involved in ADS design kit creation and model verification. The audience includes design kit developers and CAD administrators. Because the ADS Design Kit Model Verification tool is a script driven tool, the audience using this manual is expected to know some Practical Extraction Report Language (*Perl*) scripting and object oriented programming in order to perform advanced operations.

For simple circuits, you may be able to perform a verification with minimal *Perl* scripting knowledge. This document has been written to include examples for those who might not have the skills, but still want to verify simple circuits. The document provides predefined scripts that you can copy and modify for basic operation.

If you are a novice *Perl* user, review the section on "Using Perl" on page 2-1 to help you get started.

The Design Kit Model Verification Tool Process

The basic steps for using the ADS Design Kit Model Verification tool are as follows:

1. Install the default template and parameter modules.
2. Build a script header to use in your own scripts.
3. Create any custom template & parameter scripts and then add them to the default modules.
4. Create a test script and run it to simulate your test circuits.
5. If you receive errors, review the error logs, modify your scripts to correct the errors, and re-run the simulation.
6. If no errors occur, set up an ADS Data Display template to view your results.
7. Compare your simulation results between the different simulators.
8. Document your simulation results using the ADS Electronic Notebook.

Chapter 3, Building a Basic Script describes the details for performing the operations listed above. Figure 1-1 on the following page shows a simple process flowchart of the process described above.

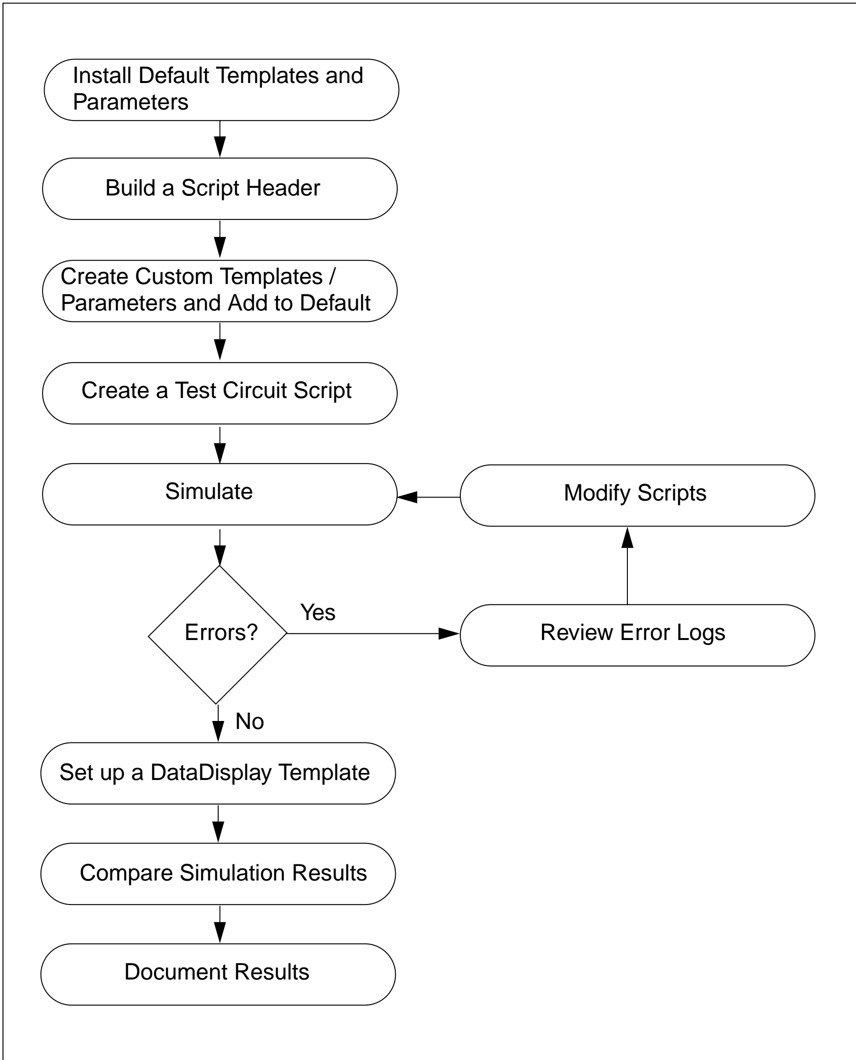


Figure 1-1. Design Kit Model Verification Process Flow

What's in this Manual

This guide contains the information needed to verify an ADS design kit model against another simulator design kit model. Most of the details are applicable for all work involving design kit model verification but some optional steps may be skipped for simpler kits or smaller installations.

- Chapter 2, Setting up the Design Kit Model Verification Tool describes the information needed to setup the design kit model verification tools such as setting up environment variables.
- Chapter 3, Building a Basic Script provides several examples that describe the steps necessary for using the design kit model verification tool. These steps are briefly described in "The Design Kit Model Verification Tool Process" on page 1-2.
- Chapter 4, Building an Advanced Script provides an example of building a more complex design kit test circuit script. The end of the chapter also provides several tables that describe the functions available in the design kit model verification tool. These tables give a description for each function as well as usage syntax.
- Chapter 5, Comparing Simulation Results describes how to setup templates for viewing your simulation results within an ADS Data Display and how to do script based comparisons.
- Chapter 6, Documenting Your Simulation Results describes the fundamentals of using the ADS Electronic Notebook.

Minimum ADS Installation Requirements

ADS 2001 or later is the minimum requirement for using the ADS Design Kit Model Verification Tool described in this manual.

Chapter 2: Setting up the Design Kit Model Verification Tool

This chapter describes the operations necessary for setting up the ADS Design Kit Model Verification Tool.

Supported Simulators

There are several simulators currently supported by the ADS Design Kit Model Verification Tool. All simulators currently supported by the verification tool are listed in Table 2-1.

Table 2-1. Design Kit Model Verification Tool Supported Simulators

Vendor	Simulator	Versions
Agilent Technologies	hpeesofsim	hpeesofsim 2001 and later
Avant! Corporation	Hspice	2000.2 and later
Cadenced Design Systems Inc.	Spectre	4.4.3/4.4.5/4.4.6

Additional simulators to those listed in Table 2-1 may be supported in future revisions of the ADS Design Kit Model Verification Tool. For more information on supported simulators, consult your Agilent Technologies sales representative.

Using Perl

Practical Extraction Report Language (*Perl*) is a scripting language widely used for system administration and programming on the World Wide Web. The Design Kit Model Verification tool makes extensive use of Perl scripts in order to provide the flexibility to write and customize your own perl scripts. The Design Kit Model Verification tool is currently script driven only and has no user interface (UI). The thought behind making this tool script driven only was that even if a UI had been implemented, most users would quickly revert to using scripts to use the tool due to its ability to be called by other perl scripts.

Perl scripts are made up of a list of statements (terminated by semicolons) and declarations. The first line of a perl script uses the **#!** symbols followed by the full path name of the directory where the version of Perl resides (see "Using Perl on UNIX" on page 2-3). When Perl detects the comment character (**#**) within the

program, it ignores the rest of the line. Many of the examples in this document use several comment characters to denote information about the lines of code following the comments. The comment character is a good tool for helping you to document your own perl scripts within the code.

Note On unix, the first 16 bits (2 bytes) of a file determine its file type. The **#!** placed as the first characters on the first line are the first 16 bits that indicate this file type is a shell script file where the script interpreter is given next. So, first having a blank line and then a line with **#!** does not work - in this case, **#!** is simply a comment.

On all systems, you run your Perl script by providing the script name as a parameter to the Perl interpreter. For example, if you had created a script called, *myPerlScript.pl*, you would enter the following command to execute the script:

```
perl myPerlScript.pl
```

Perl statements can be simple or compound statements, consisting of a variety of operators, modifiers, expressions, and functions. The ADS design kit model verification tool provides a predefined selection of functions that help you build your script. As you become more proficient in building your scripts, you will find that modification of existing scripts is a simple way to add to your design kit model verification tool set.

Note that within your perl scripts, it is required to use vertical quoted strings ('*<string>*' or *"<string>"*) wherever the chosen value happens to be a perl command, otherwise your script will not run correctly. Within the perl scripts, it is highly recommended that you quote all strings to reduce the possibility of generating errors.

Note In perl, single quotes ('*<string>*') represent text as is, with no variable substitution. Double quotes (*"<string>"*) perform variable substitution. For example, *"$myVar"* will be substituted by its value if it is in double quotes, while *'$myVar'* will just be equal to *$myVar* if it is in single quotes.

Perl originated in the UNIX community and therefore has a strong UNIX slant; however, usage within the Windows community is growing rapidly. *ActivePerl* is a port of core Perl to Windows.

For detailed information on the Perl scripting language, refer to **http://www.perl.com** on the O'Reilly & Associates, Inc. network.

The Perl scripting language is interpreted by a program called, *perl* (note the small 'p'). Advanced Design System 2002 contains a perl executable (*perl 5.6.0*) that is located under:

$HPEESOF_DIR/tools/bin (on unix)

and

$HPEESOF_DIR/toos/perl/bin (on PC)

Using Perl on UNIX

Typically, the Perl program is run by either making it directly executable, or by passing the name of the source file as an argument in the command line. When possible, it is beneficial for both */usr/bin/perl* and */usr/local/bin/perl* to be symbolic links to the actual binary file. If this is not possible, system administrators are strongly encouraged to add symbolic links to *perl* and its accompanying utilities into a directory typically found along the user's PATH, or some other convenient location.

Note In the design kit model verification tools perl script, *#!/usr/bin/perl* is used in the first line of the program. Either the actual Perl binary file or a symbolic link should be in place.

Using Perl on Windows

When using the *ActiveState* installer for Perl, a WinNT installation modifies the Registry to associate the *.pl* extension with the perl interpreter. If you install Perl by other means (including building from the sources), you may have to modify the Registry yourself. Note that this means that you can no longer tell the difference between an executable Perl program and a Perl library file. If you do not want to modify the registry, the perl <sourcefile> command from a shell window should run as on unix if the PATH variable is correctly defined.

If the invocation of the different simulators does not succeed as planned, try to use the cygnus bash shell on the PC (located in $HPEESOF_DIR/tools/bin) for running your perl scripts (see "Configuring the Environment" on page 2-4).

Note When running a perl command you may get the error, *can't open perl script "/<path2file>": No such file or directory.* This is because your current directory is not on the same drive as the perl file you want to access. To solve this you must run the command as perl *<drive_letter>*:/*<path2file>*.

Configuring the Environment

Upon installation, the verification tool attempts to find all files in the directory tree pointed to by an environment variable called DKITVERIFICATION. All executable files are located in the $DKITVERIFICATION/bin directory. If you are using the standard installation, you can add *dKitVrun* before any verification tool command which will set-up the verification tool specific environment correctly so the command will run. For example:

```
dKitVrun dKitTemplateCreate.pl
```

The *dKitVrun* command is located in $HPEESOF_DIR/bin.

To avoid using *dKitVrun*, you can setup the environment variables yourself. When the verification tool is installed, the dkitverification install directory is $HPEESOF_DIR/design_kit/verification.

Note $HPEESOF_DIR is the location of the ADS installation (see "Developing the Script Header" on page 3-2).

To ease the use of the verification tool, add the declaration of this variable to the initialization scripts.

For UNIX:

Append the following to your,

~/.profile if you are using the Korn shell:

export DKITVERIFICATION=*<dkitverification install directory>*
export PATH=$PATH:$DKITVERIFICATION/bin

Or

~/.cshrc if you are using the C shell:

setenv DKITVERIFICATION <*dkitverification install directory*>
setenv PATH ${PATH}:$DKITVERIFICATION/bin

After adding the environment variables to the initialization file, either re-logon or source your initialization file using one of the following commands:

. ~/.profile		for the Korn shell

Or

source ~/.cshrc		for the C-shell

For the PC:

Set the system wide Environment variable

DKITVERIFICATION <*dkitverification install directory*>

and add

DKITVERIFICATION\bin

to the PATH using the **Start > Settings > Control Panel**, System tool.

On some PC operating systems, it might be necessary to directly run the perl scripts from within a *Cygnus* shell. To configure the shell:

1. Start a cygnus shell ($HPEESOF_DIR/tools/bin/bash.exe).
2. Issue the following commands to mount the necessary bin directories:

```
mount $HPEESOF_DIR/tools/bin /bin
mount $HPEESOF_DIR/tools/bin /usr/bin
```

Modifying the Source Code

If you have a need to modify the source code, you must copy the entire verification tree to another directory (e.g. $HPEESOF_DIR/custom/verification), and set the DKITVERIFICATION environment variable accordingly. If you copy the entire verification tree to another directory, you will not be able to use the *dKitVrun* utility as described in "Configuring the Environment" on page 2-4 unless you edit it and change the paths accordingly.

Prior to Running a Script

Before attempting to develop a perl script and run the verification tool as described in Chapter 3, Building a Basic Script or Chapter 4, Building an Advanced Script, it is

important to ensure that your environment has been configured properly and all of the default parameter and template modules have been created.

To quickly setup a new work directory for running the design kit model verification tool, create a new directory and from within that directory, issue the following command.

```
dKitSetupWork.pl
```

Also, verify that your PATH has been modified so that the *hpeesofsim*, *spectre* and *hspice* simulators can be found and the correct environment variables have been set for them to work. For more information on configuring your environment for these simulators, consult the appropriate simulator installation documentation.

Chapter 3: Building a Basic Script

This chapter describes how to setup and build a simple script to create a netlist and run a simulation using the design kit model verification tools. A tutorial example is provided that will help you understand the fundamental requirements for verifying a simple circuit.

The example commands given should run as is on supported unix systems or from within a *cygnus* shell on a PC, provided that the environment (see “Configuring the Environment” on page 2-4) and script headers (see “Developing the Script Header” on page 3-2) are correctly configured.

Note Within the perl scripts, it is required to use vertical quoted strings wherever the chosen value happens to be a perl command, otherwise your script will not run correctly. Within the perl scripts, it is highly recommended that you quote all strings to reduce the possibility of generating errors. Open quotes (‘*<myString>*’) are not allowed for quoting strings in Perl and vertical quotes ('*<myString>*') should be used instead. Perl uses the open quote character for other applications.

The three main requirements to create and run a design kit model verification script are:

- “Understanding Templates” on page 3-4
- “Understanding Parameters” on page 3-8
- “Creating a Test Circuit” on page 3-10

Understanding Items and Handles

A netlist is a succession of lines describing the test circuit. Each element (instance, analysis, parameter, etc.) of a circuit is referred to as an *item* in the design kit model verification tool. Generally speaking, each *item* corresponds to one line in a netlist file. In the design kit model verification tool, the context of an item has been generalized. For instance, dKit circuits, dKit parameters and dKit templates are also items. Therefore, the complete test circuit is also considered an item.

To create a netlist with the design kit model verification tool, a perl script is needed. Within the perl script, items are created by means of the call to a **new()** function. The return value of that function is called a *handle*. The implementation of the functions

used in the design kit model verification tool reside in the design kit modules. Hence you get the following syntax:

$<handle> = <design_kit_module>->new(<arguments>);

For example, in perl syntax:

```
$myResistor = dKitInstance->new('R', 'R1');
```

In the example above, *$myResistor* is a handle to the item *R1*, (which is currently a non-connected resistor with an undefined resistor value).

Another example, in perl syntax:

```
$myCircuit = dKitCircuit->new("test1");
```

In this example, *$myCircuit* is a handle to an empty circuit, called *test1*.

Subsequent operations on this handle would then complete the definition of the item. For example:

```
$myCircuit->addInstance($myResistor);
```

would define *$myResistor (R1)* to actually belong to *$myCircuit (test1).*

Yet another example:

```
$myResistor->parameterValue('resistance', 10);
```

defines *$myResistor* to be a 10 Ohm resistor.

For detailed information on using items and handles, refer to Chapter 4, Building an Advanced Script.

Developing the Script Header

The first step in creating your own perl script is to build a script header that will ensure that the correct modules are found by the script.

To begin developing your perl script header:

1. Open any ASCII text editor and copy the lines of code shown in Table 3-1 to the beginning of your new file.

Table 3-1. Perl Script Header

```
#!<myPath2perl>/perl

BEGIN {
    if ($ENV{DKITVERIFICATION} eq "")
    {
         if ($ENV{HPEESOF_DIR} eq "")
        {
             print "\nPlease set environment variable DKITVERIFICATION\nto
point to the design kit model verification installation directory\n\n";
             exit 1;
         } else
         {
             $ENV{DKITVERIFICATION} =
"$ENV{HPEESOF_DIR}/design_kit/verification";
         }
    }
    $myLibPath = "$ENV{DKITVERIFICATION}/perl/lib";
    if ( ! -d $myLibPath)
    {
         if ($ENV{DKITVERIFICATION} eq
"$ENV{HPEESOF_DIR}/design_kit/verification")
         {
             if ( ! -d "$ENV{HPEESOF_DIR}/design_kit/verification")
             {
                  print "\nERROR: Unable to find verification module
directory\nVerification tool not installed at default\nlocation
\$HPEESOF_DIR/design_kit/verification\n";
                  print "Please set environment variable DKITVERIFICATION to
point\nto the design kit model verification installation directory\n\n";
             } else
             {
                  print "\nERROR: Unable to find verification module directory
at\ndefault location \$HPEESOF_DIR/design_kit/verification/perl/lib\n";
                  print "Please set environment variable DKITVERIFICATION\nto
point to the installation directory\n\n";
             }
         } else
         {
                 print "\nERROR : Unable to find verification module directory
\$DKITVERIFICATION/perl/lib\n\n";
                 print "Please set environment variable DKITVERIFICATION\nto
point to the design kit model verification installation directory\n\n";
         }
         exit 1;
    }
# To find the standard supplied libraries in the local path
    use Cwd;
    $curDir = cwd;
}

use lib "$myLibPath";
use lib "$curDir";
```

2. Replace the first line in the file header in Table 3-1 with the path to the perl interpreter. The default path is '/usr/bin/perl'; however, your path may be different.
3. Save this new file as *header.pl* so it can be copied and re-used in later scripts.

A sample script is included in the design kit model verification tool, at the location $HPEESOF_DIR/design_kit/verification/perl/header.pl

Understanding Templates

In a previous section, "Understanding Items and Handles" on page 3-1, it was shown that a *test circuit* in the design kit model verification tool consists of *handles* to *items*. When the netlist for the simulator needs to be created, the design kit model verification tool needs to know how to do this. This is where templates are used. A template, as such, can be seen as the set of rules for how to translate the *handle* in the perl test circuit script to a netlist segment for each of the different simulators. This means that one template is required for each type of component/analysis/... used in the test circuit. Different instances of the same component/analysis will use the same template.

Templates are made available to the design kit model verification tool by means of a perl module called, *dKitTemplate.pm*. So if you do not see the file *dKitTemplate.pm* within your working directory, you have not installed any templates.

To view which templates are currently available:

- Run the perl script *dKitTemplateList.pl*. This script will list all currently defined templates, with the enumeration of their nodes, parameters and simulator translation rules.
- Alternatively, you can just look at the *dKitTemplates.pm* file if you are a perl expert.

Installing the Default Templates

For standard components (R, L, C, M, Q, etc.), circuit parameters and standard analyses (DC, AC, Noise and S-parameter), the design kit model verification tool distribution includes a perl script that contains the standard template definitions. To install the standard templates, move to your working directory and issue the command:

```
dKitTemplateCreate.pl
```

This command creates the file *dKitTemplate.pm* with a set of standard templates for your design in your working directory. Once created, the *dKitTemplate.pm* file makes these templates available to be called by your test circuit perl script.

Creating Custom Templates

This section provides a step-by-step example for creating a simple custom component template. A custom component is any component that is not part of the standard component set provided with Advanced Design System. More in particular, a custom component is one that is using a model defined within the model library file. For custom components, the templates need to be created separately.

To define a custom component template:

1. Open a copy of the *header.pl* script that was created in "Developing the Script Header" on page 3-2.
2. Add the following line to the end of the *header.pl* script to initialize your custom template.

   ```
   use dKitTemplate;
   ```

3. Start defining your custom component template by creating a handle for the template structure. For example, adding the following command will create a new design kit template with a handle of *my_2p*.

   ```
   $my_2p = dKitTemplate->new('my_2p');
   ```

   If the handle you created for your template already exists, a *template already defined* error will be reported when you try to re-run your script. To avoid template handle duplication errors, run the default design kit template creation script again before re-running your script. For more information on the default design kit template creation script, refer to "Installing the Default Templates" on page 3-4.

   Alternatively, you can query to see if a template already exist before defining a new one, by adding the following lines to your script instead of just defining a handle.

   ```
   if ( ! ($my_2p = dKitTemplate->getTemplate('my_2p')) )
      {
           $my_2p = dKitTemplate->new('my_2p')
      } else
      {
   ```

```
        print "redefining my_2p\n";
    }
```

Adding the code above will result in the *my_2p* handle being redefined if another instance of *my_2p* already exists. For more information on handles, refer to "Understanding Items and Handles" on page 3-1.

4. Once you have designated a handle for your template, you can start defining your template dialects. The following example shows a simple template using the handle created earlier in step 3.

```
$my_2p->netlistInstanceTemplate('ads', '<model>:#instanceName %n1 %n2
[[count=<count>]]');

$my_2p->netlistInstanceTemplate('spectre', '#instanceName %n1 %n2
<model> [[count=<count>]]');

$my_2p->netlistInstanceTemplate('hspice', '#instanceName %n1 %n2
<model> [[count=<count>]]');
```

The example above shows the itemType *my_2p* defined to be a component with two nodes *n1*, and *n2* (see %n1, %n2), and 2 parameters, *model* and *count* (see <model>, <count>). The [[...]] syntax means that the parameter *count* is optional. The *#instanceName* is an internal function which will return the name of the item.

This template could for instance be used to define a component with 2 nodes whose model name is <model> and has a parameter called <count>, defined in the model library files of the different simulators.

Note The number of nodes and the name of the parameters used by the template is the number of nodes defined and the names of the parameters in the model library file. The number of nodes and the names of the parameters defined in the design kit customization files are ignored by the design kit model verification tool. The design kit customization files are the .ael files, which define the components (items) to be used on the schematic (item definition).

For more information on the template module, and a more rigorous explanation of the syntax, refer to "Template Module" on page 4-3.

5. Once you have completed all of your template definitions, add the following line as the last line of your test script:

```
dKitTemplate->createTemplateModule;
```

This command will add your custom template definition to the default templates.

6. Save your new script as *myTemplates.pl* or some other appropriate name. An example of what your new script might look like is shown in Table 3-2.
7. Run the new perl script to add your custom templates to the default templates in the file *dKitTemplate.pm*. You can now use your custom templates when creating your design kit test circuit.

Table 3-2. Perl Script with a call to *dKitTemplate*

```
#!/usr/bin/perl

# ADD SCRIPT HEADER from Table 3-1.

### Initialize the new template.
use dKitTemplate;

### Query to check for template duplication errors.
if ( ! ($my_2p = dKitTemplate->getTemplate('my_2p')) )
   {
        $my_2p = dKitTemplate->new('my_2p')
   } else
   {
        print "redefining my_2p\n";
   }

### Set the template dialects.
$my_2p->netlistInstanceTemplate('ads', '<model>:#instanceName %n1 %n2 [[count=<count>]]');
$my_2p->netlistInstanceTemplate('spectre', '#instanceName %n1 %n2 <model> [[count=<count>]]');
$my_2p->netlistInstanceTemplate('hspice', '#instanceName %n1 %n2 <model> [[count=<count>]]');

### Add the template definition to the default template.
dKitTemplate->createTemplateModule;

### end of script
```

Understanding Parameters

The parameters of devices in different simulators use different names. Ideally, you want the parameters of devices in your results file to have the same name. The design kit model verification tool includes a parameter module to map the different parameter names for the same parameter to one unique name.

Installing the Default Parameters

Again, there are default parameters defined. To install them into your working directory, enter the command:

```
dKitParameterCreate.pl
```

Creating Custom Parameters

Once your default parameters are installed, you can start creating your new design kit parameters in a perl file.

To create your new parameter file:

1. Open a copy of the *header.pl* script that was created in "Developing the Script Header" on page 3-2.
2. Add the following line to the end of the *header.pl* script to initialize your custom template.

   ```
   use dKitParameter;
   ```

3. To define a custom parameter, you must create an instance and define the different instance parameters. For example:

   ```
   $IDC = dKitParameter->new('Idc'); # The dKit Parameter Name
   $IDC->description("dc current"); # Its description
   $IDC->dialectReference('hspice', 'dc'); # Its name in hspice
   $IDC->dialectReference('ads', 'Idc'); # Its name in ads
   $IDC->dialectReference('spectre', 'dc'); # Its name in spectre
   $IDC->resultName("dc"); # The name used for the results
   ```

   The dKit Parameter Name is the name to use for device parameters within the test circuit perl script. The netlister will translate it to the simulator parameter name. For more information, refer to "Parameter Module" on page 4-9.

4. As with the templates, it is beneficial to add new parameter definitions to the default parameter definitions using the command:

```
dKitParameter->createParameterModule;
```

Your new script should now appear similar to Table 3-3. Run the script after entering this command to update the parameter information.

Table 3-3. Perl Script for Custom Parameters

```
#!/usr/bin/perl

# ADD SCRIPT HEADER from Table 3-1.

use dKitParameter;
### add the perl commands to create the parameters here.
$IDC = dKitParameter->new('Idc'); # the dKit Parameter Name
$IDC->description("dc current"); # its description
$IDC->dialectReference('hspice', 'dc'); # its name in hspice
$IDC->dialectReference('ads', 'Idc'); # its name in ads
$IDC->dialectReference('spectre', 'dc'); # its name in spectre
$IDC->resultName("dc"); # the name used for the results
dKitParameter->createParameterModule;
### end of script
```

Creating a Test Circuit

Once your templates and parameters are defined and installed, you can start creating your design kit test circuit in a new perl file. This section provides examples for creating a simple test circuit.

The following example illustrates many of the issues typically encountered when building a test circuit. Each of the sections shown below discusses an aspect of creating the test circuit perl script. While all portions of the creation process may not be used in your script, each section should be addressed in the order given.

1. "Setting up a Test Circuit" on page 3-11
2. "Adding Circuit Options" on page 3-12
3. "Adding Model Library Files" on page 3-13
4. "Creating Instances" on page 3-14
5. "Connecting Instances" on page 3-15
6. "Creating Analyses" on page 3-15
7. "Defining the Simulators" on page 3-18
8. "Running the Test Circuit Script" on page 3-22

Setting up a Test Circuit

To create a new test circuit perl script:

1. Open a copy of the *header.pl* script that was created in "Developing the Script Header" on page 3-2.
2. Add the following line to the end of the *header.pl* script. This line is used to initialize your new test circuit.

   ```
   use dKitCircuit;
   ```

3. Create the circuit handle for the structure by adding the following function:

   ```
   $Circuit = dKitCircuit->new('<circuitName>');
   ```

   Where *<circuitName>* determines where to store the intermediate and final results. If the name contains slashes (/), subdirectories are created for storing the intermediate results and the final result (*<circuitName>*.cti or *<circuitName>*_*<simulator>*.cti) in CITI file format.

   The dataset used to view the results in the ADS Data Display will be placed in the current directory. The ADS Data Display file name is in the form *<circuitName>*.ds or *<circuitName>*_*<simulator>*.ds where the slashes (/) are replaced by underscores (_). For example, if *<circuitName>* equals *myTests/test1* then all intermediate files are stored in the directory *myTests*, and will start with the prefix *test1*. The resulting dataset will be called myTests_test1.ds or myTests_test1_*<simulator>*.ds. The _*<simulator>* is added to the final results name if you request a simulation for only one simulator.

   For more information on datasets (.ds) and CITI file (.cti) format, refer to Chapter 20 of the ADS *"Circuit Simulation"* documentation.

4. Save a copy of your new file as *myCircuit.pl*.

Adding Items

The content of your circuit is defined by adding items as needed to your script using the **new()** function.

To create a new item, use the function in the general form:

```
new('<templateName>', '<itemName>');
```

Each *<itemName>* should be a unique identifier. For more information on items, refer to "Understanding Items and Handles" on page 3-1.

Note Item names are case sensitive. Therefore, *Optional1* is not equal to *optional1*. Also, because Hspice translates everything to lower case in the results file, comparisons will be easier when only lower case itemNames are used.

Adding Circuit Options

The Options component makes it possible to set general simulation options related to convergence tolerances, warnings, global noise temperature and so on. To set an Option other than the standard circuit Options:

1. Create the Option handle.
2. Define the simulator parameter values.
3. Add the simulator option to the circuit.

The following example shows a typical simulator options setup.

```
$OPTION1 = dKitInstance->new('SIMULATOROPTION', "option1");
$OPTION1->parameterValue('adsOptions', "ResourceUsage=yes");
$OPTION1->parameterValue('spectreOptions', "save=lvlpub digits=10");
$OPTION1->parameterValue('hspiceOptions', "nopage acct=0 dccap=1
numdgt=10");
$Circuit->addSimulatorOption($OPTION1);
```

Temperature is handled somewhat differently than other options. For example, to set the circuit temperature at 20 degrees Celsius, you need the following:

```
$TEMP = dKitInstance->new('TEMPERATURE', 'T1');
$TEMP->parameterValue('temperature', 20);
$Circuit->addSimulatorOption($Temp)'
```

If you plan to do a temperature sweep, it is better to not add a default temperature option, as this might cause problems when doing Hspice simulations.

You can add multiple option instances; however, ensure that you give each option a unique name. For example, *option1*, *option2*, *option3, etc*.

Adding Model Library Files

The modelLibrary item enables you to build a list of model files that you want to include in your test circuit.

To add a modelLibrary item to your circuit:

1. Create a modelLibrary handle.
2. Add the path to the variables and equations data.
3. Set the path and Section parameters if needed (see "Adding a Model Library Section Designator" on page 3-13).
4. Add the modelLibrary to the circuit.

For each additional library or library Section, you must create and add a new model library handle. The following example shows a typical modelLibrary item setup.

```
$MODLIB1 = dKitInstance->new('MODELLIBRARY', "vars");
$MODLIB1->parameterValue('adsModelLibrary', "../models/ads/vars.net");
$MODLIB1->parameterValue('spectreModelLibrary',"../models/spectre/vars.sc
s");
$MODLIB1->parameterValue('hspiceModelLibrary',"../models/hspice/vars.hsp"
);
$Circuit->addModelLibrary($MODLIB1);
```

Note When using relative paths, please note that the simulator actually changes the working directory to the *<circuitName>* directory before it starts the simulator. For more information, refer to "Setting up a Test Circuit" on page 3-11.

Adding a Model Library Section Designator

Each model file can have a *Section* designator. This enables you to include only a portion of a model file for corner analysis, provided your model file has been set up properly. The Section designator is optional; if it is left empty, the entire file will be included (provided it has no dependencies on needing a particular Section set up).

When a model library has different sections and the ADS model library file is adapted using *#define* statements, a library Section can be selected. For example, to select the section NORMAL:

```
$MODLIB1->parameterValue('adsSectionName', "NORMAL");
$MODLIB1->parameterValue('hspiceSectionName', "NORMAL");
$MODLIB1->parameterValue('spectreSectionName', "NORMAL");
```

Creating Instances

An instance in a design kit is typically a component definition that generally has similar netlist formats for each of the different simulators.

To start creating instances:

1. Create the instance handle.
2. Define the instance parameters.
3. Add the instance to the circuit.

For example, to create a circuit with two 1 volt DC voltage sources (V1 and V2) and a 10 ohm resistor (R1):

```
$V1 = dKitInstance->new('V', 'V1');
$V1->parameterValue('Vdc', 1);
$Circuit->addInstance($V1);

$V2 = dKitInstance->new('V', 'V2');
$V2->parameterValue('Vdc', 1);
$Circuit->addInstance($V2);

$R1=dKitInstance->new('R', 'R1');
$R1->parameterValue('resistance', 10);
$Circuit->addInstance($R1);
```

Note that the parameter values should be in engineering notation if they are real numbers, and should not contain units. For example, a capacitor of 1nF would use:

```
$C1=dKitInstance->new('C', 'C1');
$C1->parameterValue('capacitance', 1e-9);
$Circuit->addInstance($C1);
```

To know which parameters are defined for which instances refer to “Defining and Retrieving the Parameter Values” on page 4-22.

Connecting Instances

A circuit node is a connection of the terminals or pins between two or more instances. After all instances are added, you can connect them to each other using the **nodeName()** function supplied in the design kit model verification tool.

For example, to connect the internal node/pin/terminal *nplus* of Vds, the internal node/pin/terminal *n1* of R1:

```
$Vds->nodeName('nplus', 'node1');
$R1->nodeName('n1', 'node1');
```

Where *node1* is the name of the circuit node or connection point.

If you are working on a layout, you would probably use *'trace1'* in place of *'node1'*, where *'trace1'* would be the name of the trace connecting your components. For the design kit model verification tool, both nodes and traces are equivalent. However, the circuit node convention is generally used in the examples provided in this document.

For more information on node names for instances, refer to “Defining and Retrieving Node Names” on page 4-23.

Creating Analyses

The pure analyses that are currently supported by the design kit model verification tool are:

- AC (ac analysis)
- DC (dc analysis)
- SP (S-parameter analysis)
- NOISE (noise analysis)

All supported pure analyses require one SWEEP parameter to be defined and consequently a SWEEPPLAN. If the default output from the simulators is not adequate, an OUTPUTPLAN can be added to these analyses to enumerate the desired outputs. If more parameters need to be swept, a SWEEP analysis can be added as well.

Note OUTPUTPLANs added to a S-parameter analysis will be ignored.

The following example shows how to set up a DC analysis with two swept parameters; the DC voltage of the voltage source V1 and the temperature.

1. In this example, there are two parameters that must be swept. Therefore, two analysis handles are required.

   ```
   $SW1 = dKitAnalysis->new('SWEEP', 'SWEEP1');
   $DC1 = dKitAnalysis->new('DC', 'DC1');
   ```

2. The analyses need to be cascaded with the Sweep analysis in front. To ensure that the Sweep analysis is first, the analysis is stacked using the **addSubAnalysis()** function.

   ```
   $SW1->addSubAnalysis($DC1);
   ```

3. The defined Sweep parameters chosen in this example use the SWEEP analysis to sweep the DC voltage of V1, and the DC analysis to sweep the temperature.

   ```
   $SW1->parameterValue('device', 'V1');
   $SW1->parameterValue('parameter', 'Vdc');
   $DC1->parameterValue('parameter', 'temp');
   ```

   The sweep definition could be reversed if needed. For *global* parameters such as *temp* and *freq*, you do not need to specify the device.

4. To setup the SweepPlans for simulation results for:

   - Fifty-one voltages of V1.
   - Vdc equally spaced between 0 to 2.5V.
   - Temperatures of -40, 27, and 155 degrees Celsius

   The following SWEEPPLANs are defined:

   ```
   $SWEEPPLAN1 = dKitAnalysis->new('SWEEPPLAN_LIN', 'SWL1');
   $SWEEPPLAN1->parameterValue('start', 0);
   $SWEEPPLAN1->parameterValue('stop', 2.5);
   $SWEEPPLAN1->parameterValue('numPts', 51);
   $SW1->addSweepPlan($SWEEPPLAN1);

   $SWEEPPLAN2 = dKitAnalysis->new('SWEEPPLAN_PT', 'SWP1');
   $SWEEPPLAN2->parameterValue('values', "-40 27 155");
   $DC1->addSweepPlan($SWEEPPLAN2);
   ```

5. If you defined a DC or AC analysis, you should also add an OutputPlan; otherwise there will not be any output from Hspice.

Note ADS and Spectre both generate a default OutputPlan if no plan is specified. For Noise and S-parameter analysis, a default outputplan is always added.

The most common output request is the value of the voltage at a certain node or trace. For this type of OutputPlan, you can use the OUTPUTPLAN_NODES default template. Using this template, you specify the nodes for which the voltage is desired as the value to the *'nodes'* parameter. For example:

```
$OUTPUTNODES = dKitAnalysis->new('OUTPUTPLAN_NODES', "out1");
$OUTPUTNODES->parameterValue('nodes', "node1 node2");
$DC1->addOutputPlan($OUTPUTNODES);
```

To output the currents in a certain device, the default template OUTPUTPLAN_CURRENTS is provided. Using this template, you can specify the terminals of the devices for which you want the currents.

Note Currently ADS does not allow any selection of currents. However, ADS outputs the current through *terminal 1* of voltage sources by default.

```
$OUTPUTCURRENTS=dKitAnalysis->new('OUTPUTPLAN_CURRENTS', "out2");
$OUTPUTCURRENTS->parametervalue('deviceTerminals', "V1:1 V2:1");
$DC1->addOutputPlan($OUTPUTCURRENTS);
```

For DC only, the device operating points (DOP) can be saved using the default OUTPUTPLAN_DOPS template. Using this template, you can specify the desired operating points as the value of its *deviceParameters* parameter. The names used to specify the parameters are the names of the design kit parameters (see “Understanding Parameters” on page 3-8).

```
$OUTPUTDOPS=dKitAnalysis->new('OUTPUTPLAN_DOPS', "out3");
$OUTPUTDOPS->parameterValue('deviceParameters', "V2:i");
$DC1->addOutputPlan($OUTPUTDOPS);
```

Note The ADS simulator only allows you to select whether or not to save device operating points. So for ADS, you will get all device operating points or none.

6. To add the defined circuit analysis to the test circuit, use the **addAnalysis()** function as shown in this example.

```
$Circuit->addAnalysis($SW1);
```

Defining the Simulators

To define the simulators to run for ADS, Spectre, and Hspice, add the line:

```
$Circuit->simulate('ads', 'spectre', 'hspice');
```

Your new test circuit script should now appear similar to the "Example Test Circuit Script" on page 3-19.

Example Test Circuit Script

```
#!/usr/bin/perl
BEGIN {
    if ($ENV{DKITVERIFICATION} eq "")
    {
         if ($ENV{HPEESOF_DIR} eq "")
        {
             print "\nPlease set environment variable DKITVERIFICATION\nto
point to the design kit model verification installation directory\n\n";
             exit 1;
         } else
         {
             $ENV{DKITVERIFICATION} =
"$ENV{HPEESOF_DIR}/design_kit/verification";
         }
    }
    $myLibPath = "$ENV{DKITVERIFICATION}/perl/lib";
    if ( ! -d $myLibPath)
    {
         if ($ENV{DKITVERIFICATION} eq
"$ENV{HPEESOF_DIR}/design_kit/verification")
         {
             if ( ! -d "$ENV{HPEESOF_DIR}/design_kit/verification")
             {
                  print "\nERROR: Unable to find verification module
directory\nVerification tool not installed at default\nlocation
\$HPEESOF_DIR/design_kit/verification\n";
                  print "Please set environment variable DKITVERIFICATION to
point\nto the design kit model verification installation directory\n\n";
             } else
             {
                 print "\nERROR: Unable to find verification module directory
at\ndefault location \$HPEESOF_DIR/design_kit/verification/perl/lib\n";
                 print "Please set environment variable DKITVERIFICATION\nto
point to the installation directory\n\n";
             }
         } else
         {
                print "\nERROR : Unable to find verification module directory
\$DKITVERIFICATION/perl/lib\n\n";
                 print "Please set environment variable DKITVERIFICATION\nto
point to the design kit model verification installation directory\n\n";
         }
         exit 1;
    }
# To find the standard supplied libraries in the local path
    use Cwd;
```

```
    $curDir = cwd;
}

use lib "$myLibPath";
use lib "$curDir";

### Initialize the new test circuit.
use dKitCircuit;
$Circuit = dKitCircuit->new('myCircuit');

### Set Simulator Options.
$OPTION1 = dKitInstance->new('SIMULATOROPTION', "option1");
$OPTION1->parameterValue('adsOptions', "ResourceUsage=yes");
$OPTION1->parameterValue('spectreOptions', "save=lvlpub digits=10");
$OPTION1->parameterValue('hspiceOptions', "nopage acct=0 dccap=1
numdgt=10");
$Circuit->addSimulatorOption($OPTION1);

### Add Model Files.
### Note that model libraries are not needed here because default components
### are used.These lines have been commented out.
#$MODLIB1 = dKitInstance->new('MODELLIBRARY', "vars");
#$MODLIB1->parameterValue('adsModelLibrary', "../models/ads/vars.net");
#$MODLIB1->parameterValue('spectreModelLibrary',"../models/spectre/vars.scs"
);
#$MODLIB1->parameterValue('hspiceModelLibrary',"../models/hspice/vars.hsp");
#$Circuit->addModelLibrary($MODLIB1);

### Set the Library Section.
#$MODLIB1->parameterValue('adsSectionName', "NORMAL");
#$MODLIB1->parameterValue('hspiceSectionName', "NORMAL");
#$MODLIB1->parameterValue('spectreSectionName', "NORMAL");

### Create Instances and Define Their Parameters.
$V1 = dKitInstance->new('V', 'V1');
$V1->parameterValue('Vdc', 1);
$Circuit->addInstance($V1);

$V2 = dKitInstance->new('V', 'V2');
$V2->parameterValue('Vdc', 1);
$Circuit->addInstance($V2);

$R1=dKitInstance->new('R', 'R1');
$R1->parameterValue('resistance', 10);
$Circuit->addInstance($R1);

### Connect the Nodes.
$V1->nodeName('nminus', 0);
```

```
$V1->nodeName('nplus', 'node1');
$R1->nodeName('n1', 'node1');
$R1->nodeName('n2', 'node2');
$V2->nodeName('nplus', 'node2');
$V2->nodeName('nminus', 0);

### Create Analysis Handles.
$SW1 = dKitAnalysis->new('SWEEP', 'SWEEP1');
$DC1 = dKitAnalysis->new('DC', 'DC1');

### Stack the Analyses.
$SW1->addSubAnalysis($DC1);

### Define the SWEEP Parameters.
$SW1->parameterValue('device', 'V1');
$SW1->parameterValue('parameter', 'Vdc');
$DC1->parameterValue('parameter', 'temp');

### Define and Add the SWEEPPLANS.
$SWEEPPLAN1 = dKitAnalysis->new('SWEEPPLAN_LIN', 'SWL1');
$SWEEPPLAN1->parameterValue('start', 0);
$SWEEPPLAN1->parameterValue('stop', 2.5);
$SWEEPPLAN1->parameterValue('numPts', 51);
$SW1->addSweepPlan($SWEEPPLAN1);

$SWEEPPLAN2 = dKitAnalysis->new('SWEEPPLAN_PT', 'SWP1');
$SWEEPPLAN2->parameterValue('values', "-40 27 155");
$DC1->addSweepPlan($SWEEPPLAN2);

### Add the OUTPUTPLANS.
$OUTPUTNODES = dKitAnalysis->new('OUTPUTPLAN_NODES', "out1");
$OUTPUTNODES->parameterValue('nodes', "node1 node2");
$DC1->addOutputPlan($OUTPUTNODES);

$OUTPUTCURRENTS=dKitAnalysis->new('OUTPUTPLAN_CURRENTS', "out2");
$OUTPUTCURRENTS->parameterValue('deviceTerminals', "V1:1 V2:1");
$DC1->addOutputPlan($OUTPUTCURRENTS);

$OUTPUTDOPS=dKitAnalysis->new('OUTPUTPLAN_DOPS', "out3");
$OUTPUTDOPS->parameterValue('deviceParameters', "V2:i");
$DC1->addOutputPlan($OUTPUTDOPS);

### Add the Top Analysis to the circuit.
$Circuit->addAnalysis($SW1);

### Add the Simulation Command.
$Circuit->simulate('ads', 'spectre', 'hspice');
```

```
### List any missing parameter definitions.
print dKitParameter->listMissingParameterDefinitions;

### End of script
```

Running the Test Circuit Script

If you did not initialize the template or parameter module yet, i.e. you do not have a *dKitTemplate.pm* or *dKitParameter.pm* file in your working directory, you need to do it now.

To initialize the template module:

```
dKitTemplateCreate.pl
```

To initialize the Parameter module:

```
dKitParameterCreate.pl
```

Alternatively, both of the above modules can be initialized at the same time using:

```
dKitSetupWork.pl
```

If you run the "Example Test Circuit Script" on page 3-19, your results should appear in a dataset (*<name>*.ds) format.

You may receive several warning messages after running your script.

```
Please define parameter <param> using ...
```

This is to inform you that the dKit specific parameter *<param>* could not be translated to a simulator specific parameter, but kept its dKit Specific name. You may also have gotten the warning:

```
Please define a parameter using ... so that the <dialect> parameter
<param> ...
```

This means that the *<dialect>* simulator specific parameter *<param>* could not be translated to a result name, but kept its simulator specific name. In this case you can freely choose the name of the dKit parameter.

If you want to list an overview of which parameters were not translated to a result name, add the following line to the end of your TEST CIRCUIT script after the line `$Circuit->simulate('ads', 'spectre', 'hspice');`

```
print dKitParameter->listMissingParameterDefinitions;
```

Viewing Your Results in a Data Display

Simulation results are stored in *datasets* (*.ds) and *citifiles* (*.cti). To view your results in an ADS Data Display:

1. Launch ADS and create a new project *<adsproject>* if necessary.
2. Copy all of your datasets to your ADS project's *data* directory (*<adsproject>*/data).
3. Launch the ADS Data Display server from your ADS window. From the ADS *Main* window, choose **Window > New Data Display**.
4. Create a template to view your results. For more information, refer to *"Using a Template in Your Display"* in the ADS *"Data Display"* documentation.
5. Select the dataset to view the results. If your parameters have been defined correctly, you should be able to use the *<dataset>.<simulator>.<parameter>* to access the different results for the different simulators, with *<simulator>* equal to ads_sim, spectre_sim or hspice_sim.

For more information on *datasets* (*.ds) and *citifiles* (*.cti), refer to Chapter 20 of the ADS *"Circuit Simulation"* manual.

For more information on simulation results, refer to Chapter 5, Comparing Simulation Results.

Chapter 4: Building an Advanced Script

This chapter first gives a more thorough description (function level) of the different modules which make up the design kit model verification tool. This information is required before you can proceed with creating more advanced scripts.

Module Reference

The perl module files (*.pm) contain the functions used for creating the individual perl scripts. The different modules used by the design kit model verification tool are shown here.

- "Template Module" on page 4-3
- "Parameter Module" on page 4-9
- "Circuit Module" on page 4-14
- "Instance and Analysis Modules" on page 4-21
- "Results Module" on page 4-27

For a summary of all template functions provided in the perl modules and their basic functionality, refer to "Design Kit Model Verification Tool Perl Modules" on page 4-32.

Using the Module Debug Facility

All modules have a debug facility which can be turned on using the command:

```
dKit<module>->debug(<level>);
```

A *<level>* value of 0 turns debugging off. The higher the value of *<level>*, the more information you will receive. The information you receive will depend on the design kit *<module>* you are debugging.

The value of *<module>* can be Template, Parameter, Instance, Analysis or Circuit.

E.g. dKitCircuit->debug(1); will not delete the intermediate files (e.g netlist files).

Currently, the highest debug value is 100; however, not all intermediate values are used.

Quoting your string-variables

Within the perl scripts, it is required to use vertical quoted strings ('*<string>*' or *"<string>"*) wherever the chosen value happens to be a perl command, otherwise your script will not run correctly. Within the perl scripts, it is highly recommended that you quote all strings to reduce the possibility of generating errors.

Note In perl, single quotes ('*<string>*') represent text as is, with no variable substitution. Double quotes (*"<string>"*) perform variable substitution. For example, *"$myVar"* will be substituted by its value if it is in double quotes, while *'$myVar'* will just be equal to *$myVar* if it is in single quotes.

Open quotes (‘*<string>*’ or “*<string>*”) are not allowed for quoting strings in perl and vertical quotes ('*<string>*' or "*<string>*") should be used instead. While there is a very subtle difference in the appearance of the two styles, perl will not accept open quotes.

Template Module

A template can be considered as the set of rules for how to translate the item defined using the dKit* commands to the simulator netlist dialect (See "Understanding Templates" on page 3-4).

The template module (*dKitTemplate.pm*) contains functions to perform the following operations:

- "Creating a New Template Handle" on page 4-4
- "Retrieving a Handle of an Existing Template" on page 4-4
- "Documenting a Template" on page 4-4
- "Retrieving the Template Documentation" on page 4-4
- "Setting the Item Name Prefix" on page 4-4
- "Defining the Translation Rules" on page 4-5
- "Defining Template Parameters and Nodes/Pins" on page 4-7
- "Setting a Default Template Parameter Value" on page 4-7
- "Retrieving the Default Template Parameter Value" on page 4-7
- "Creating a Template for a Subcircuit" on page 4-7
- "Displaying the Defined Templates" on page 4-8
- "Saving the Template Definitions to Allow Fast Access" on page 4-8

Creating a New Template Handle

To create a handle to a new template, use the function:

```
$Template = dKitTemplate->new('<templateName>');
```

Note that *<templateName>* should be a unique name. If a template is redefined using an existing template name, an error will be generated.

Retrieving a Handle of an Existing Template

To retrieve a handle of an existing template use the function:

```
$Template = dKitTemplate->getTemplate('<templateName>');
```

$Template is undefined if the template with *<templateName>* does not exist.

Documenting a Template

To document your template, use the function:

```
$Template->description("<templateDescriptionText>");
```

Where *<templateDescriptionText>* is the text describing the template.

Retrieving the Template Documentation

To get the documentation of a template, use the function:

```
$myDescription = $Template->description();
```

$myDescription will then contain the string describing the template.

Setting the Item Name Prefix

Hspice requires that the name of instances of certain components start with a predefined prefix. This is also implemented in the dKit verification syntax by the function:

```
$Template->requiredPrefix("<prefix>");
```

For example, Hspice uses the prefix “r” to designate a resistor. Therefore, defining the “r” prefix for a resistor (whose Template Handle is $R) would be done by:

```
$R->requiredPrefix("r");
```

The prefixes are case insensitive, this means that in the example above, resistor names can start with R or r.

Note All names are lowercase in the Hspice results file. If you require an Hspice simulation and you want your results dataset to reflect the simulator name, your devices/nodes should only contain lower case. For more information, refer to "Viewing Your Results in a Data Display" on page 3-23.

Defining the Translation Rules

To define how your template will be translated for each dialect, use the *netlistInstanceTemplate* function:

```
$Template->netlistInstanceTemplate("<dialect>", "<netlistFormat>");
```

where *<dialect>* is the supported simulator dialect and *<netlistFormat>* is the defined structure for the netlist. An example for a standard resistor is shown in Table 4-1.

Table 4-1. Example Template for a Resistor

```
$R = dKitTemplate->new('R');
$R->description("Resistor");
$R->requiredPrefix('r');

$R->netlistInstanceTemplate('hspice', '#instanceName %n1 %n2 <modelName> R=<resistance>
[[TC1=<tempCoef1>]] [[TC2=<tempCoef2>]] [[SCALE=<scale>]] [[M=<multiplicity>]] [[AC=<acResistance>]]
[[DTEMP=<dtemp>]] [[L=<length>]] [[W=<width>]] [[C=<capacitanceN2Bulk>]]');

$R->netlistInstanceTemplate('ads', 'R:#instanceName %n1 %n2 R=<resistance> [[Temp=<temp>]]
[[Tnom=<tnom>]] [[TC1=<tempCoef1>]] [[TC2=<tempCoef2>]] [[Noise=<noise>]] [[M=<multiplicity>]]');

$R->netlistInstanceTemplate('spectre', '#instanceName %n1 %n2 resistor r=<resistance>
[[trise=<dtemp>]] [[tc1=<tempCoef1>]] [[tc2=<tempCoef2>]] [[isnoisy=<noise>]]
[[m=<multiplicity>]]');
```

When defining the netlist format, the following special directives are available:

- **#function**: Calls the internally defined function or hook. Currently, **#instanceName** is the only function supported. **#instanceName** returns the name of the item. The other **#names** you will find in the *dKitTemplateCreate.pl* file are hooks, i.e. they direct the netlist function of the Circuit Module to follow a different path in the code. So you need to be a perl expert to modify those within the perl modules of the design kit model verification tool distribution. For

information on modifying the source code, refer to "Modifying the Source Code" on page 2-5.

- %node: Adds the internal node/pin node to this item, e.g. %n1
- *<name>*: Adds name as a template parameter. If you assign a value to this parameter when defining the item that uses this template, *<name>* will be replaced by its value in the netlist.
- ~eval(text)end: Means that the text, after all parameters have been substituted by their values, will be passed to the perl *eval* function. In this way you can do operations on the parameters before they are substituted in the netlist.

  For example, observe the following netlist definition.

```
$T->netlistInstanceTemplate(ads, 'TLIN4:#instanceName %in %refin %out
%refout [[Z=<Z0>]] [[F=<frequency>]] [[E=~eval(<electricalLength> *
360;)end]]');
```

  The ADS parameter E equals 360 times the design kit parameter electricalLength. This is a simple example, actually you can include whole perl scripts. If you need additional variables in the text, be sure to use the *my declarator*, otherwise it will not work! This is a perl specific declarator to say variables are local.

  The previous example with a more explicit definition would be:

```
$T->netlistInstanceTemplate(ads, 'TLIN4:#instanceName %in %refin %out
%refout [[Z=<Z0>]] [[F=<frequency>]] [[E=~eval(my $electricalLength
="<electricalLength>"; if ($electricalLength ne "") {return
$electricalLength* 360;} else {return "";})end]]');
```

- [[test]]: Means what is between the [[ ]] is optional, and will not be netlisted for the item using this netlist unless all parameters which are listed between the [[ ]] and are not part of an *~eval(text)end* expression are defined AND all top-level *~eval(text)end* evaluations return non-empty strings.

  If you do not put a parameter definition between [[ ]] then this parameter is required, and you will get an error if you try to netlist an item using this template which does not have a value defined for this parameter. For example, the resistance shown in the resistor example of Table 4-1 uses R=<resistance>.

  If an *~eval(text)end* expression is not between [[]] then this expression is required, and you will get an error if the top-level ~eval(text)end evaluation

returns an empty string. It is not an error for a nested ~eval(text)end evaluation to return an empty string.

You can find more examples of templates in the *dKitTemplateCreate.pl* file.

Defining Template Parameters and Nodes/Pins

While parsing the translation rules, the internal nodes/pins and parameters are automatically added to the template. If you want to add additional nodes and parameters, you can use the functions:

```
$Template->addNode("nodeName");
$Template->addParameter("parameterName");
```

Setting a Default Template Parameter Value

A parameter of a template can be given a default value, which will be netlisted if the item using this template does not redefine the parameter. By giving a parameter a default value, you make it a required parameter, which will be netlisted always.

You assign a default value by means of the function

```
$Template->parameterValue("<parameter>", "<value>");
```

The parameter needs to be defined otherwise this function will return an error.

Retrieving the Default Template Parameter Value

You retrieve the default template parameter value using the function:

```
$myDefaultValue = $Template->parameterValue('<parameter>');
```

If no default value is assigned, an empty string is returned.

Creating a Template for a Subcircuit

If you have defined a circuit WITH ONLY INSTANCES using the functions in *dKitCircuit.pm* (see below) you can convert it to a template using the function:

```
$Template = dKitTemplate->createSubcircuit(
        "<templateName>",
        $circuit,
        <nodelist>, # eg [n1, n2, n3]
        <requiredParameterList>, # eg [l, w]
```

```
        <optionalParameterList>); # eg []
```

If a new simulator has to be added, the hash table %subcircuitVocabularyHash in the file *dKitTemplate.pm* has to be modified for this function to work correctly. In this table:

INSTANCETEMPLATE defines how the template will be netlisted when it is netlisted as an instance in the main circuit.

HEADERTEMPLATE defines the command which the simulator uses to start a subcircuit definition within the netlist

FOOTERTEMPLATE defines the line which the simulator uses to terminate the definition of a subcircuit within a netlist

The strings &nodes, &templateName, &requiredParameters, &optionalParameters, used within these definitions, will be replaced by the values given to the subroutine.

Displaying the Defined Templates

If you want to display which templates are currently defined, you can use one of the following functions at any time with the perl template definition script:

```
dKitTemplate->dumpTemplates;
```

or

```
print dKitTemplate->templates2perl;
```

Saving the Template Definitions to Allow Fast Access

For faster access, it is better to have your templates stored in a hash table than having to read them from a file. Therefore, use the function,

```
dKitTemplate->createTemplateModule;
```

to translate your template definitions into a hash table and add them to the module *dKitTemplate.pm* in your working directory.

It is a good practice to keep the definition of your templates in a separate file. Optimally - without being annoyed by the error *template already defined* - use the example shown in Table 4-2 for constructing the template definition perl script file.

Table 4-2. Example Template Definition Script

```
#!<myPath2Perl>/perl

# ADD SCRIPT HEADER from Table 3-1.

use dKitTemplate;

if (! ($my_2p = dKitTemplate->getTemplate(my_2p)))
{ # define your template here
    $my_2p = dKitTemplate->new(my_2p);
    $my_2p->requiredPrefix("X");
    $my_2p->netlistInstanceTemplate(ads, '<model>:#instanceName %n1 %n2 [[count=<count>]]
[[xrsub=<xrsub>]] [[xctrench=<xctrench>]] [[w=<w>]] [[l=<l>]] [[rreq=<rreq>]] [[nk=<nk>]]
[[na=<na>]] [[ng=<ng>]] [[ctcfg=<ctcfg>]] [[dgeo=<dgeo>]] [[sgeo=<sgeo>]]');
    $my_2p->netlistInstanceTemplate(spectre, '#instanceName %n1 %n2 <model> [[count=<count>]]
[[xrsub=<xrsub>]] [[xctrench=<xctrench>]] [[w=<w>]] [[l=<l>]] [[rreq=<rreq>]] [[nk=<nk>]]
[[na=<na>]] [[ng=<ng>]] [[ctcfg=<ctcfg>]] [[dgeo=<dgeo>]] [[sgeo=<sgeo>]]');
    $my_2p->netlistInstanceTemplate(hspice, '#instanceName %n1 %n2 <model> [[count=<count>]]
[[xrsub=<xrsub>]] [[xctrench=<xctrench>]] [[w=<w>]] [[l=<l>]] [[rreq=<rreq>]] [[nk=<nk>]]
[[na=<na>]] [[ng=<ng>]] [[ctcfg=<ctcfg>]] [[dgeo=<dgeo>]] [[sgeo=<sgeo>]]');
}

# add other templates definitions here


dKitTemplate->createTemplateModule;

### end of template creation script file
```

Parameter Module

The parameter module is used to map different variable names used by the different simulators for the same parameter. For example, the dc voltage of a voltage source is called:

- **Vdc** for ADS
- **dc** for Spectre and Hspice.

To avoid conflicts between variable names, you can set up parameter mappings that define the parameter name (name used in the design kit model verification tool), the dialect name (the name used by the simulator), and the result name (the name used when processing the results). See "Creating a Parameter Handle" on page 4-11.

There are several dialects currently supported by the design kit model verification tool. For information on supported dialects, refer to "Supported Simulators" on page 2-1.

The parameter module (*dKitParameter.pm*) contains functions to perform the following operations:

- "Creating a Parameter Handle" on page 4-11
- "Documenting a Parameter" on page 4-11
- "Retrieving the Parameter Documentation" on page 4-11
- "Defining the Dialect Name for a Parameter" on page 4-11
- "Defining the Result Name for a Parameter" on page 4-12
- "Displaying the Defined Parameters" on page 4-12
- "Saving the Parameter Definitions to Allow Fast Access" on page 4-13
- "List All Missing Parameter Definitions" on page 4-13

Creating a Parameter Handle

To create a handle to a new parameter, use the function:

```
$Parameter = dKitParameter->new( '<parameterName>' );
```

Where *<parameterName>* is the name to be used in the verification tool perl scripts.

Example:

Using the following example to create a parameter which will define the dc voltage of a voltage source:

```
$VDC = dKitParameter->new('Vdc');
```

you will have to use "Vdc" as the value of the "parameter" parameter of a Sweep Analysis if you want to sweep the dc voltage of a voltage source.

Documenting a Parameter

To document your template, use the function:

```
$Parameter->description( "<parameterDescriptionText>" );
```

Where *<parameterDescriptionText>* is the text describing the parameter

Example:

```
$VDC->description("DC Voltage");
```

Retrieving the Parameter Documentation

To retrieve the documentation of a parameter, use the function:

```
$myDescription= $Parameter->description;
```

Defining the Dialect Name for a Parameter

The different dialect names for the parameter are added by using the function:

```
$Parameter ->dialectReference( '<dialect>' , '<dialectParameterName>');
```

Example:

```
$VDC->dialectReference('spectre', 'dc');
$VDC->dialectReference('hspice', 'dc');
$VDC->dialectReference('ads', 'Vdc');
```

Defining the Result Name for a Parameter

To define the result name for the parameter use the function:

```
$Parameter->resultName(" <resultName> ");
```

Because Hspice result files use lowercase, it is advised to use all lowercase resultNames if you want to make comparisons with Hspice results.

Note The *resultName* names should be chosen in order that *$parameter($dialect)->resultName* is unique. For example, $VDC(hspice)->dc.

Examples

```
$VDC = dKitParameter->new("Vdc");
$IDC = dKitParameter->new("Idc");
$IDC->dialectReference('hspice', 'dc');
$VDC->dialectReference('hspice', 'dc');
$IDC->dialectReference('ads', 'Idc');
$VDC->dialectReference('ads', 'Vdc');
```

Using the following *resultName*s

```
$IDC->resultName("idc");
$VDC->resultName("vdc");
```

would create ambiguity, because both have the Hspice reference 'dc'. I.e. in the results file you will find dc, but you do not know whether it stems from IDC being translated to DC or VDS being translated to DC. Instead, use:

```
$IDC->resultName("dc");
$VDC->resultName("dc");
```

Displaying the Defined Parameters

To display the parameters which are currently defined you can use one of the following functions at any time with the perl parameter definition script

```
dKitParameter->dumpDKitXrefs;
```

or

```
print dKitParameter->dKitXrefs2perl;
```

Saving the Parameter Definitions to Allow Fast Access

For faster access, it is better to have your parameters stored in a hash table than having to read them from a file. Therefore, use the function:

```
dKitParameter->createParameterModule;
```

to translate your parameter definitions into a hash table and add them to the module *dKitParameter.pm* in your working directory.

List All Missing Parameter Definitions

To list all missing parameter definitions, use the following function after the circuit is simulated:

```
dKitParameter->listMissingParameterDefinitions;
```

Circuit Module

To initialize the circuit module, add the following line before using any functions to define your test circuit.

```
use dKitCircuit;
```

The circuit module (*dKitCircuit.pm*) contains functions to perform the following operations:

- “Creating a New Circuit Handle” on page 4-15
- “Setting the Circuit Name” on page 4-15
- “Adding Simulator Options” on page 4-15
- “Removing Simulator Options” on page 4-15
- “Adding a Path to a Model Library” on page 4-16
- “Removing a Path to a Model Library” on page 4-16
- “Adding Components/Instances” on page 4-16
- “Removing Components/Instances” on page 4-17
- “Adding Analyses” on page 4-17
- “Removing Analyses” on page 4-17
- “Simulating a Circuit” on page 4-17

For the more advanced users, additional functions are available to perform the following operations:

- “Adding a Startdeck and Enddeck Card” on page 4-18
- “Defining and Modifying SubCircuit Parameters” on page 4-18
- “Creating the Simulator Netlist” on page 4-18
- “Simulating a Circuit from a Netlist” on page 4-19
- “Freeing Memory Occupied by the Simulation Results” on page 4-19

Creating a New Circuit Handle

To create a new circuit, use the function:

```
$Circuit = dKitCircuit->new("<circuitName>");
```

See also "Setting the Circuit Name" on page 4-15.

Setting the Circuit Name

You set the circuitName using the function:

```
$Circuit->circuitName("<circuitName>");
```

Where *<circuitName>* determines where all data will be stored. If *<circuitName>* contains forward slashes (/), all intermediate files, including the simulator netlists, will be placed in a subdirectory. See also "Simulating a Circuit" on page 4-17.

Adding Simulator Options

There are some default options given for the different simulators if you do not specify any options. However, if you specify one option for a given simulator, all other default options will be erased.

The following options MUST be set for the verification tool to run correctly:

```
spectre : rawfmt=psfascii
ADS : UseNutmegFormat=no ASCII_Rawfile=yes
```

and therefore they will ALWAYS be added to the netlist.

To add simulator options, use the function:

```
$Circuit->addSimulatorOption($OPTION);
```

where $OPTION is a reference to an instance. See "Instance and Analysis Modules" on page 4-21.

Removing Simulator Options

To remove a previously added simulator option from your circuit, use the function:

```
$Circuit->removeSimulatorOption($Instance);
```

Adding a Path to a Model Library

To add the path to a modelLibrary, use the function:

```
$Circuit->addModelLibary($MODELLIBRARY);
```

where $MODELLIBRARY is a reference to an instance. See "Instance and Analysis Modules" on page 4-21.

Removing a Path to a Model Library

To remove a previously added path to a modelLibrary from your circuit, use the function:

```
$Circuit->removeModelLibrary($Instance);
```

Adding Components/Instances

To add components or netlist parameters, use the function:

```
$Circuit->addInstance($Instance);
```

where $Instance is a reference to an instance. See "Instance and Analysis Modules" on page 4-21.

The order in which the instances are added will also be the order in which the instances will be netlisted.

This is for instance important for PARAMETER instances, because in Hspice parameter instances defining an equation should only be netlisted (added) after the elements or parameters within the equation are defined.

If you have the following definition:

```
$P1 = dKitInstance->new('PARAMETER', 'rsq');
$P1->parameterValue('value', 1);

$P2 = dKitInstance->new('PARAMETER', 'r_l');
$P2->parameterValue('value', 'rsq*r_w');

$P3 = dKitInstance->new(PARAMETER, 'r_w');
$P3->parameterValue('value',1.34e-6);
```

then, `$P2` should be added (using `$Circuit->addInstance`) AFTER `$P1` AND `$P3` have been added.

Removing Components/Instances

To remove a previously added instance from your circuit, use the function:

```
$Circuit->removeInstance($Instance);
```

Adding Analyses

To add analyses, use the function:

```
$Circuit->addAnalysis($Analysis);
```

where $Analysis is a reference to an analysis. See instance/analysis module.

Note Currently only one analysis can be added to a circuit.

Removing Analyses

To remove a previously added analysis from your circuit, use the function:

```
$Circuit->removeAnalysis($Instance);
```

Simulating a Circuit

To simulate a circuit after it is defined, issue the command

```
$Circuit->simulate("<dialect1>", "<dialect2>", "<dialect3>", "...");
```

The currently supported dialects are defined in “Supported Simulators” on page 2-1.

The output of the simulation function creates a dataset (*<circuitNameForDs>*.ds) in your working directory. The *<circuitNameForDs>* is the name you give to your circuit using the function:

```
$Circuit->circuitName("<circuitName>");
```

where all forward slashes (/) are replaced by underscores (_).

It also creates a citi file (*<circuitNameForCti>*.cti). This will be placed in the correct subdirectories if *<circuitNameForCti>* contains forward slashes (/).

If you only supply one dialect to the simulate function, _<dialect> will be appended to the <circuitName> when creating <circuitNameForCti> and <circuitNameForDs>.

All intermediate files, including the simulator netlists, will be removed unless you set the circuit level debugging to '1' or higher using the debug command:

```
dKitCircuit->debug(<level>);
```

Adding a Startdeck and Enddeck Card

There is also a possibility to add a *startDeckCard* and *endDackCard*. This is done by changing the defaults for the template CIRCUITSTARTDECKCARD and CIRCUITENDDECKCARD respectively. By default a Spectre file will start with 'simulator lang=spectre', and Hspice will end a file with .end.

Defining and Modifying SubCircuit Parameters

For subcircuits it might be necessary to modify circuit parameters. To get the value of a circuitParameter, use:

```
$Circuit->parameterValue("<parameterName>");
```

To set the value of a circuitParameter, use:

```
$Circuit->parameterValue("<parameterName>", "<value>");
```

The parameter needs to be defined otherwise you will get an error. To define a parameter, you can use the function:

```
$Circuit->addParameter("<parameterName>");
```

Normally you will not need this function because parameters are automatically created from the nelistTemplate.

Creating the Simulator Netlist

Normally the netlist is automatically created when you simulate your test circuit. However, if you want to have access to the netlist without doing a simulation, you can use the command:

```
$myNetlist = $Circuit->netlist("<dialect>");
```

which will assign the netlist to the perl variable *$myNetlist*, or:

```
$Circuit->netlist("<dialect>", "<fileName>");
```

which will write the netlist to the file *<fileName>*. If you want to simulate the netlist after editing, refer to “Simulating a Circuit from a Netlist” on page 4-19 for information on choosing the fileName.

Note If the *<fileName>* is equal to the *<circuitName>* with the default dialect extension, that file will be removed if the same perl script also performs a simulation and is using circuit debug level 0.

Simulating a Circuit from a Netlist

To run a simulation given a netlist file, issue the command:

```
dKitCircuit->simulate("<netlist1>", "<netlist2>", ...);
```

where <netlist1>, <netlist2> are the names of the netlist files. The suffixes of the netlists will determine which simulator will be called. They are defined by the parameters *<dialect>*NetlistSuffix of the default Template CIRCUITNETLISTNAME. They are ".hsp" for Hspice, ".ckt" for the ADS circuit simulator, ".scs" for Spectre. If the basenames (i.e. the name without the suffix) are equal a combined dataset will be created, just like if you would have used the command:

```
$Circuit->simulate("<dialect1>", "<dialect2>", ... );
```

See “Simulating a Circuit” on page 4-17.

Freeing Memory Occupied by the Simulation Results

When the simulation completes, the simulation results are normally kept in memory in order to have fast access to the results when additional statistical operations need to be performed. However, for large circuits the amount of data kept might be substantial and when simulating a lot of circuits from within the same perl file you might encounter an *out of memory error* during the execution of the perl test circuit script. To prevent this from happening you can issue the command:

```
$Circuit->deleteResultsFromMemory();
```

when you do not need the simulation results any more for the circuit. It will free up the memory taken by the results so it can be used by other simulations.

Note Do not change the name of the circuit between the call to simulate and *deleteResultsFromMemory*, or the *deleteResultsFromMemory* function will have no effect.

Instance and Analysis Modules

Instances and Analyses are very similar (they are both defined using templates); however, they are put into separate modules because:

- Instances (components/options/...) have a very similar netlist format for the different simulators, Analyses do not.
- Analyses can include subanalyses, Instances cannot include subinstances. A subcircuit is just an instance.

The following operations can be performed by both modules:

- "Creating a New Item Handle" on page 4-22
- "Viewing the Template Definition" on page 4-22
- "Defining and Retrieving the Parameter Values" on page 4-22

The instance module (*dKitInstance.pm*) contains functions to perform the following instance specific operations:

- "Retrieving the InstanceName" on page 4-23
- "Defining and Retrieving Node Names" on page 4-23

The analysis module (*dKitAnalysis.pm*) contains functions to perform the following analysis specific operations:

- "Retrieving the AnalysisName" on page 4-23
- "Adding a Sweepplan" on page 4-23
- "Removing a Sweepplan" on page 4-24
- "Adding an OutputPlan" on page 4-24
- "Removing an OutputPlan" on page 4-24
- "Adding a Sub Analysis" on page 4-24
- "Removing a Sub Analysis" on page 4-25
- "Scale/Offset a Sweep Parameter" on page 4-25

Creating a New Item Handle

An instance is created using the command:

```
$InstanceHandle = dKitInstance->new("templateName", "instanceName");
```

Note that the same name cannot be used for two different instances.

An analysis is created using the command:

```
$AnalysisHandle = dKitAnalysis->new("templateName", "analysisName");
```

Note that the same name cannot be used for two different analyses.

For a list of standard supported instances and analysis, refer to Appendix A, Template List.

Viewing the Template Definition

To retrieve a description of the template definition use:

```
$myTemplateDef = $InstanceOrAnalysisHandle->template->dump2str;
```

To print it to the screen use the standard perl print function, e.g.

```
print "$myTemplateDef\n";
```

This will cause the template definition to be printed to the terminal window upon completion of the script.

Alternatively you could use the method described in “Displaying the Defined Templates” on page 4-8, or the Template reference table in Appendix A, Template List if it is a standard template.

Defining and Retrieving the Parameter Values

Once an instance or analysis is created, the value of the parameters which are defined by the template need to be set. To set a parameter value, use the command:

```
$InstanceOrAnalysisHandle->parameterValue("parameterName", "value");
```

If value is numeric, it should be in engineering or floating notation (1e-3 or 0.001) without units. For example, the value for the capacitance of a 1nF capacitor should be specified as 1e-9.

To retrieve the value of a parameter use:

```
$myValue = $InstanceOrAnalysisHandle->parameterValue("parameterName");
```

To know which parameters are available, see "Viewing the Template Definition" on page 4-22.

Retrieving the InstanceName

The instanceName of an instance can be retrieved using:

```
$myName = $InstanceHandle->instanceName;
```

Defining and Retrieving Node Names

If an instance has nodes, which is the case for all components, then you must define the node names using:

```
$Instance->nodeName("internalName", "circuitNodeName");
```

This assigns the name of the circuitNode to the internal instance node. For more information, refer to "Connecting Instances" on page 3-15.

To retrieve the name of the circuitNode use the command:

```
$myCircuitNode = $Instance->nodeName("internalName");
```

For information on knowing which internal nodes are defined, refer to "Viewing the Template Definition" on page 4-22.

Retrieving the AnalysisName

The analysisName of an analysis can be retrieved using:

```
$myName = $AnalysisHandle->analysisName;
```

Adding a Sweepplan

To add a sweepplan to an analysis use:

```
$AnalysisHandle->addSweepPlan($SweepPlanHandle);
```

Note Currently only one sweepplan/analysis is supported.

Sweepplans are a special type of analyses. They are created using:

```
$SweepplanHandle = dKitAnalysis->new("sweepplanTemplateName",
"sweepplanName");
```

For a list of standard supported sweepplans, refer to "Sweepplan Templates" on page A-6.

Removing a Sweepplan

To remove a sweepplan from an analysis use:

```
$AnalysisHandle->removeSweepPlan($SweepplanHandle);
```

Adding an OutputPlan

To add an outputplan to an analysis use:

```
$AnalysisHandle->addOutputPlan($OutputplanHandle);
```

Note Multiple outputplans can be added to an analysis.

Outputplans are a special type of analyses. They are created using:

```
$OutputplanHandle = dKitAnalysis->new("outputplanTemplateName",
"outputplanName");
```

For a list of standard supported outputplans, refer to "Outputplan Templates" on page A-4.

Removing an OutputPlan

To remove an outputplan from an analysis use:

```
$AnalysisHandle->removeOutputPlan($OutputplanHandle);
```

Adding a Sub Analysis

To add a subanalysis to a sweep analysis use:

```
$AnalysisHandle->addSubAnalysis($SubAnalysisHandle);
```

Note Currently only one subanalysis / analysis is supported.

Removing a Sub Analysis

To remove a subanalysis from a sweep analysis use:

```
$AnalysisHandle->removeSubAnalysis($SubAnalysisHandle);
```

Scale/Offset a Sweep Parameter

There is no specific analysis module function to scale/offset a specific sweep parameter, but the sweepplan templates have parameters <dialect>Scale and <dialect>Offset that are intended for this purpose.

This functionality is for instance needed when you want to do a sweep of the DC voltage of a Port instance. In Spectre the DC-voltage is multiplied by 2, while this is not the case in ADS or Hspice.

So to do a sweep of the dc voltage of a port instance your perl script could look like:

```
$P1=dKitInstance->new('PORT', 'V1');
$P1->parameterValue('Vdc', 0.7);
$SWEEPPLAN = dKitAnalysis->new('SWEEPPLAN_LIN', 'Plan1');
$SWEEPPLAN->parameterValue('start', 0.5);
$SWEEPPLAN->parameterValue('stop', 1.5);
$SWEEPPLAN->parameterValue('numPts', 5);
$SWEEPPLAN->parameterValue('spectreScale', 0.5);
$SP1= dKitAnalysis->new('SP', 'Sp');
$SP->parameterValue('device', 'V1');
$SP->parameterValue('parameter', 'Vdc');
$SP->addSweepPlan($SWEEPPLAN);
```

To get the original sweep values in the result file (see “Results Module” on page 4-27) you should also add to your script:

```
dKitResults->sweepVariableScale("V1.dc", "spectre", 2.0);
```

Note The resultName of the parameter should be used to define the sweepvariable not the dKit Parameter name.

Results Module

If the perl script you are using is not initialized using:

```
use dKitCircuit;
```

then add the following line before using any functions of the results module:

```
use dKitResults;
```

The results module (*dKitResults.pm*) contains functions to perform the following operations:

- "Set/Get the Offset of a Sweepvariable when Processing the Simulation Results" on page 4-28
- "Set/Get the Scale of a Sweepvariable when Processing the Simulation Results" on page 4-28
- "Reading the Simulation Results or Citifiles" on page 4-29
- "Retrieving a Handle to the Results" on page 4-29
- "Writing the Results to Citifiles" on page 4-29
- "Converting Citifiles to Datasets" on page 4-30
- "Comparing Two Simulator Results" on page 4-30
- "Performing a Simple Statistical Analysis" on page 4-31
- "Merging Citifiles into One Big Citifile" on page 4-31
- "Freeing the Memory taken by the Simulation Results" on page 4-31

Set/Get the Offset of a Sweepvariable when Processing the Simulation Results

To set the offset value used when the results module processes the data for a certain parameter use:

```
$offset = dKitResults->sweepVariableOffset('parameter', 'dialect',
offsetValue);
```

Where:

`'parameter'` is in the format *<deviceName>.<parameterName>*. *<parameterName>* should be the same as the *<resultName>* of the parameter defined using the Parameter Module. `'parameter'` is also the name found in the citi file containg the results.

`'dialect'` designates the simulator results for which this offset should be applied.

`offsetvalue` is the value of the offset. It normally equals -sweepplanoffset. sweeplanoffset is the offset applied to the sweeplan used to sweep this variable, See "Scale/Offset a Sweep Parameter" on page 4-25.

To get the offset value, use:

```
$offset = dKitResults->sweepVariableOffset('parameter', 'dialect');
```

Set/Get the Scale of a Sweepvariable when Processing the Simulation Results

To set the scale value used when the results module processes the data for a certain parameter use:

```
dKitResults->sweepVariableScale('parameter', 'dialect', scaleValue);
```

Where:

`'parameter'` is in the format *<deviceName>.<parameterName>.* *<parameterName>* should be the same as the *<resultName>* of the parameter defined using the Parameter Module. `'parameter'` is also the name found in the citi file containing the results.

`'dialect'` designates the simulator results for which this scale should be applied.

`'scalevalue'` is the value of the scale. It normally equals 1.0/sweepplanscale. sweeplanscale is the scale applied to the sweeplan used to sweep this variable, See "Scale/Offset a Sweep Parameter" on page 4-25.

To get the scale value, use:

```
$scale = dKitResults->sweepVariableScale('parameter', 'dialect');
```

Reading the Simulation Results or Citifiles

To read the results from a simulation or a CITI file, use the function:

```
$Result = dKitResults->readData('<projectName>', '<dialect>');
```

Where *<projectName>* would normally be the circuitName of the simulated circuit. The supported dialects include the *citi*-dialect, as well as the supported simulators. In the case of the *citi*-dialect, the file *<projectName>*.cti is read and its content is stored in an item which can be accessed via the handle $Result;

Note The design kit model verification tool is not able to read merged citifiles. For more information, refer to "Merging Citifiles into One Big Citifile" on page 4-31.

Retrieving a Handle to the Results

When you have lost the handle to some result data, you can retrieve it using the function:

```
$Result = dKitResults->getResults(<projectName>, <dialect>);
```

If the result data does not exist, this function will try to retrieve it using:

```
dKitResults->readData
```

For more information, refer to "Reading the Simulation Results or Citifiles" on page 4-29.

Writing the Results to Citifiles

To create a Citifile with the results use the function:

```
$Result->writeCitifile('fileName'[, 'setName']);
```

Where `fileName` is the name of the file to be created and the optional `setName` is the name of the set to be used when referencing it in the ADS data display server. For more information, refer to "Comparing Two Simulator Results" on page 4-30. It is advised to use .cti as the filename extension, and use the dialect as the setName. In

this case you are compliant with the rules used within the design kit model verification tool.

Converting Citifiles to Datasets

To convert a citifile to a dataset, use the function:

```
dKitResults->convertCitifile2Dataset('path2CitiFile'[,'setName']);
```

Where `path2CitiFile` is the location of the citifile and `setName` is the name to be used when referencing it in the ADS data display server.

Note The name of the citifile should end with .cti and setName should not be used if a merged citifile is given.

The name of the dataset, will be the name of the citifile with all forward slashes (/) substituted by underscores (_), and the final .cti substituted by .ds.

For viewing the results with the ADS data display server, it is convenient if all data are located in one dataset. To do so, all citi data should be merged before the citifile is converted to a dataset. For more information, refer to "Merging Citifiles into One Big Citifile" on page 4-31.

Comparing Two Simulator Results

To calculate the difference between the results of two simulators, the following functions can be used:

```
$ReldiffHandle = dKitResults->compareDataRelative($Result1Handle,
$Result2Handle);

$AbsdiffHandle = dKitResults->compareDataAbsolute($Result1Handle,
$Result2Handle);
```

Where `$Result1Handle` and `$Result2Handle` are the handles to result items (see "Retrieving a Handle to the Results" on page 4-29).

`$ReldiffHandle` will reference the relative differences between the simulation results and `$AbsdiffHandle` will reference the absolute differences. If you want to display these results using the ADS data display server, first transform the results to citifiles (see "Writing the Results to Citifiles" on page 4-29), then transform the citifiles to datasets (see "Converting Citifiles to Datasets" on page 4-30).

The result handles can also be used to obtain some simple statistical information (see “Performing a Simple Statistical Analysis” on page 4-31). Both simulation results (`$Result1Handle` and `$Result2Handle`) should have the same sweepvariables, and only the differences between the matching dependent variables will be stored in the *ReldiffHandle* and/or *AbsdiffHandle*. Dependent variable names match if they have the same name (case sensitive).

Performing a Simple Statistical Analysis

To get the mean, maximum and minimum values of all dependent variables within a result item, use the command:

```
$ResultHandle->calculateStatistics(['outputFileName']);
```

If `'outputFileName'` is specified, the mean, maximum and minimum of all dependent variables will be written into the file `'outputFileName'`. Otherwise, it will be written to the terminal window.

Merging Citifiles into One Big Citifile

To merge several citifiles into one citifile, use the command:

```
dKitResults->mergeCitifiles('outputFileName', 'inputFileName1',
'inputFileName2', ...);
```

This is useful when you want to view the results using the ADS Data Display Server, because all data are available from within one dataset if this merged citifile is converted to a dataset.

Note The citifiles to be merged should all have different set names, otherwise the resulting dataset will be corrupt.

Freeing the Memory taken by the Simulation Results

To free the memory taken by previously run simulations, issue the command:

```
dKitResults->freeMemory('circuitName' [, 'dialect']);
```

where *circuitName* is the name of the circuit and *dialect* is the name of the simulator dialect. If no *dialect* is specified, all results for *circuitName* will be deleted.

Design Kit Model Verification Tool Perl Modules

The following sections contain a summary of the design kit model verification tool Perl Module (*.pm) files that contain the functions used for creating your individual perl scripts. The tables provided can be used as a quick reference when developing your scripts. The following information is provided for each of the functions listed:

- Function name
- Function description
- Function syntax
- Variable definitions (i.e. variables listed in angled brackets < .. >)
- Function limitations
- References to examples

Template Module (dKitTemplate.pm)

The *dKitTemplate.pm* file contains the functions used for creating the design kit model verification template script.

Table 4-3. dKitTemplate.pm Functions Summary

Function	Description and Syntax
addNode()	Adds an internal node/pin. $TemplateHandle->addNode("<nodeName>");
addParameter()	Adds a parameter. $TemplateHandle->addParameter("<parameterName>");
createSubcircuit()	Creates a subcircuit. $TemplateHandle = dKitTemplate->createSubcircuit("<templateName>", $circuit, nodeList, requiredParameterList, optionalParameterList); Where: <templateName> = The name of the template. nodeList = The list of node names (e.g. ['n1', 'n2', ...]). requiredParameterList = A list with the required parameters and their initial values (e.g. ['l', 'w']). optionalParameterList = A list with the optional parameters and their initial values (e.g. []).
createTemplateModule	Saves the currently defined templates in the file dKitTemplate.pm dKitTemplate->createTemplateModule;
debug()	The module debug facility can be used for all modules. Level 0 turns debugging off. The higher the value of level, the more information you receive. The highest level value is currently 100; however, some intermediate values are not used. dKitTemplate->debug(<level>);
description()	Used to document the template. $myDescription = $TemplateHandle->description(“<templateDescriptionText>”); If <templateDescriptionText> is given, sets the template documentation, otherwise it returns the current documentation.

Table 4-3. dKitTemplate.pm Functions Summary

Function	Description and Syntax
dumpTemplates	Displays the list of currently defined templates. dKitTemplate->dumpTemplates;
getTemplate()	Retrieves a handle to an existing template. $TemplateHandle = dKitTemplate->getTemplate('<templateName>');
netlistInstanceTemplate()	Defines how template is translated for each dialect. $TemplateHandle->netlistInstanceTemplate("<dialect>", ["<netlistFormat>"]); if <netlistFormat> is given, sets the template netlist for the given dialect, otherwise it returns the current template netlist for the given dialect.
new()	Creates a new template. $TemplateHandle = dKitTemplate->new(<templateName>);
parameterValue()	Adds a default parameter value. $TemplateHandle->parameterValue("<parameter>", [<value>]); If <value> is given, sets the value of the given parameter, otherwise it returns the value of the given parameter.
requiredPrefix()	Sets a predefined prefix. Case insensitive. $TemplateHandle->requiredPrefix(["<prefix>"]); If <prefix> is given, sets the prefix value, otherwise it returns the prefix value.
templateName	Returns the name of the template $myName = $TemplateHandle->templateName;
templates2perl	Prints the list of defined templates in perl format. print dKitTemplate->templates2perl;

Parameter Module (dKitParameter.pm)

The *dKitParameter.pm* file contains the functions used for creating the design kit model verification parameters script.

Table 4-4. dKitParameter.pm Functions Summary

Function	Description and Syntax
createParameterModule	Translates parameter definitions into a hash table. dKitParameter->createParameterModule;
debug()	The module debug facility can be used for all modules. Level 0 turns debugging off. The higher the value of level, the more information you receive. The highest level value is currently 100; however, some intermediate values are not used. dKitParameter->debug(<level>);
description()	Used to document template. $ParameterHandle->description(“<parameterDescriptionText>”);
dialectReference()	Sets the parameter name for a simulator dialect. $ParameterHandle->dialectReference("<dialect>", "parameterNameForDialect");
dumpDKitXrefs	Print all currently defined parameters. dKitParameter->dumpDKitXrefs;
listMissingParameter Definitions	List an overview of which parameters were not translated to a result name. print dKitParameter->listMissingParameterDefinitions;
new()	Add a new parameter, <parameterName>, to $ParameterHandle. $ParameterHandle = dKitParameter->new(<parameterName>);
resultName()	Defines the result name for this parameter. $ParameterHandle->resultName("myResultName");

Circuit Module (dKitCircuit.pm)

The *dKitCircuit.pm* file contains the functions used for creating the design kit model verification circuit script.

Table 4-5. dKitCircuit.pm Functions Summary

Function	Description and Syntax
addAnalysis()	Adds an analysis. $CircuitHandle->addAnalysis($Analysis);
addInstance()	Adds a component. $CircuitHandle->addInstance($Instance);
addModelLibary()	Adds the path to a modelLibrary. $CircuitHandle->addModelLibary($MODELLIBRARY);
addParameter()	Adds a parameter to a subcircuit. $CircuitHandle->addParameter("<parameterName>");
addSimulatorOption()	Adds simulator options. $CircuitHandle->addSimulatorOption($OPTION);
circuitName()	Defines a circuit name. $CircuitHandle->circuitName("<circuitName>");
debug()	The module debug facility can be used for all modules. Level 0 turns debugging off. The higher the value of level, the more information you receive. The highest level value is currently 100; however, some intermediate values are not used. dKitCircuit->debug(<level>);
dKitCircuit	Initialize all modules. use dKitCircuit;
deleteResultsFromMemory()	The deleteResultsFromMemory() function releases memory taken by the simulation results of the circuit. $CircuitHandle->deleteResultsFromMemory
netlist()	Creates the netlist for a given simulator and optionally store it in a file. $myNetlist = $CircuitHandle->netlist("<dialect>" [,"fileName"]);

Table 4-5. dKitCircuit.pm Functions Summary

Function	Description and Syntax
new()	Creates a new circuit. $Circuit = dKitCircuit->new("<circuitName>");
parameterValue()	Get the value of a circuit parameter. $CircuitHandle->parameterValue("<parameterName>"); Modify the value of a circuit parameter. $CircuitHandle->parameterValue("<parameterName>", "<value>");
removeAnalysis()	Remove an analysis from the circuit. $CircuitHandle->removeAnalysis($analysisHandle);
removeInstance()	Remove an Instance/Component from the circuit. $CircuitHandle->removeInstance($instanceHandle);
removeModelLibrary()	Remove a Model Library Path from the Circuit. $CircuitHandle->removeModelLibrary($libraryHandle);
removeSimulatorOption()	Remove an option from the circuit. $CircuitHandle->removeSimulatorOption($optionHandle);
simulate()	Simulate a circuit after it is defined. $CircuitHandle->simulate("<dialect1>", "<dialect2> ", "..."); Simulate a netlist. dKitCircuit->simulate("netlist1", "netlist2", "...");

Instance Module (dKitInstance.pm)

The *dKitInstance.pm* file contains the functions used for creating the design kit model verification parameters script.

Table 4-6. dKitInstance.pm Functions Summary

Function	Description and Syntax
debug()	The module debug facility can be used for all modules. Level 0 turns debugging off. The higher the value of level, the more information you receive. The highest level value is currently 100; however, some intermediate values are not used. dKitInstance->debug(level);
instanceName	Returns the instanceName. $myInstanceName = $InstanceHandle->instanceName;
new()	Creates an instance. $InstanceHandle = dKitInstance->new("<templateName>", "<instanceName>");
nodeName()	Defines a node name. $InstanceHandle->nodeName("<internal>", "<external>");
parameterValue()	Defines a parameter value. $InstanceHandle->parameterValue("parameterName", "value");
template	Gets the handle of the template defining the instance. $myTemplate = $InstanceHandle->template

Analysis Module (dKitAnalysis.pm)

The *dKitAnalysis.pm* file contains the functions used for creating the design kit model verification parameters script.

Table 4-7. dKitAnalysis.pm Functions Summary

Function	Description and Syntax
addOutputPlan()	Add an output plan. $AnalysisHandle->addOutputPlan($OutputPlanHandle);
addSubAnalysis()	Add to the sweep analysis the reference to a subAnalysis. $AnalysisHandle->addSubAnalysis($SubAnalysisHandle);
addSweepPlan()	Add a sweep plan. $AnalysisHandle->addSweepPlan($SweepPlanHandle);
analysisName	Retrieves the name of the analysis. $myAnalysisName = $AnalysisHandle->analysisName;
debug()	The module debug facility can be used for all modules. Level 0 turns debugging off. The higher the value of level, the more information you receive. The highest level value is currently 100; however, some intermediate values are not used. dKitAnalysis->debug(level);
new()	Creates an analysis. $AnalysisHandle = dKitInstance->new("<templateName>", "<analysisName>"); <templateName> = SWEEP, DC, AC or SP <analysisName> = Examples are Sweep1, DC1, AC1, or SP1
parameterValue()	Defines a parameter value. $AnalysisHandle->parameterValue("parameterName", "value");
removeOutputPlan()	Removes an outputplan from the analysis. $AnalysisHandle->removeOutputPlan($OutputPlanHandle);
removeSubAnalysis()	Removes a subanalysis from the analysis. $AnalysisHandle->removeSubAnalysis($SubAnalysisHandle);

Table 4-7. dKitAnalysis.pm Functions Summary

Function	Description and Syntax
removeSweepPlan()	Removes a sweepplan from the analysis. $AnalysisHandle->removeSweepPlan($SweepPlanHandle);
template	Retrieves the handle to the template definition. $myTemplate = $AnalysisHandle->template;

Results Module (dKitResults.pm)

The *dKitResults.pm* file contains the functions used for creating the design kit model verification simulation results data.

Table 4-8. dKitResults.pm Functions Summary

Function	Description and Syntax
calculateStatistics()	Calculate min, max and mean for the results. my $statistics = $ResultHandle->calculateStatistics(['outputFileName']);
compareDataAbsolute()	Compare two simulator results, and store absolute differences in new object. $AbsdiffHandle = dKitResults->compareDataAbsolute($Result1Handle, $Result2Handle);
compareDataRelative()	Compare two simulator results, and store relative differences in new object. $ReldiffHandle = dKitResults->compareDataRelative($Result1Handle, $Result2Handle);
convertCitifile2Dataset()	Convert a citifile to a dataset. dKitResults->convertCitifile2Dataset('path2CitiFile'[,'setName']);
freeMemory()	Releases the memory taken by the specified simulation results. dKitResults->freeMemory('circuitName' [, 'dialect']); where circuitName is the name of the circuit and dialect is the name of the simulator dialect. If no dialect is specified, all results for circuitName will be deleted.
getResults()	Get the simulator result data, retrieve old data if they exist. $ResultHandle = dKitResults->getResults("projectName", "dialect");
mergeCitifiles()	Merge the given inputfiles into one file. dKitResults->mergeCitiFiles("resultFileName", "inputFileName1", "inputFileName2", ...)
readData()	Read in the simulator result data, do not try to retrieve old data. $ResultHandle = dKitResults->readData('projectName', 'dialect');

Table 4-8. dKitResults.pm Functions Summary

Function	Description and Syntax
sweepVariableOffset()	Set offset for a sweepparameter while processing results for dialect. $offset = dKitResults->sweepVariableOffset('sweepparameter', 'dialect' [, offsetValue]);
sweepVariableScale()	Set scale for a sweepparameter while processing results for dialect. $scale = dKitResults->sweepVariableScale('sweepparameter', 'dialect' [, scaleValue]);
writeCitifile()	Save results in a citifile. $ResultHandle->writeCitifile('fileName'[, 'setName']);

Example Device Test Scripts

The example scripts shown in this section are for reference only. In order to use these scripts, the appropriate model files need to be provided. This involves making changes to the *diode_test_dc.pl* script to reference the required model files and to designate the correct sections. The *create_diode.pl* script may also need to be changed to reflect the correct instance line definition for the associated models.

diode_test_dc.pl

The *diode_test_dc.pl* script is a simulation to compare ADS vs. Hspice using the design kit model verification tool. It is a simple diode in parallel with a DC voltage source. The only variable swept is the forward diode voltage, *Vd*. The output is intended to be Capacitance and Current vs. Vd.

Note There are required perl scripts, *diode_create.pl* and *diode_parameters.pl* that must be executed before this script. This is so that a model with required parameters can be created for a diode. The *diode_create.pl* and *diode_parameters.pl* scripts only need to be executed once. For more details, refer to the contents of "diode_create.pl" on page 4-46 and "diode_parameter.pl" on page 4-48.

This test is written for the forward region (Vdc = 0 to 0.7) for the ndiode and pdiode diodes.

```
#!/usr/bin/perl

BEGIN {
    if ($ENV{DKITVERIFICATION} eq "")
    {
        if ($ENV{HPEESOF_DIR} eq "")
        {
            print "\nPlease set environment variable DKITVERIFICATION\nto point to the design kit
verification installation directory\n\n";
            exit 1;
        } else
        {
            $ENV{DKITVERIFICATION} = "$ENV{HPEESOF_DIR}/design_kit/verification";
        }
    }
    $myLibPath = "$ENV{DKITVERIFICATION}/perl/lib";
    if ( ! -d $myLibPath)
    {
        if ($ENV{DKITVERIFICATION} eq "$ENV{HPEESOF_DIR}/design_kit/verification")
        {
            if ( ! -d "$ENV{HPEESOF_DIR}/design_kit/verification")
            {
```

```
                print "\nERROR: Unable to find verification module directory\nVerification tool not
installed at default\nlocation \$HPEESOF_DIR/design_kit/verification\n";
                print "Please set environment variable DKITVERIFICATION to point \nto the design kit
verification installation directory\n\n";
            } else
            {
                print "\nERROR: Unable to find verification module directory at\ndefault location
\$HPEESOF_DIR/design_kit/verification/perl/lib\n";
                print "Please set environment variable DKITVERIFICATION\nto point to the installation
directory\n\n";
            }
        } else
        {
            print "\nERROR : Unable to find verification module directory
\$DKITVERIFICATION/perl/lib\n\n";
            print "Please set environment variable DKITVERIFICATION\nto point to the design kit
verification installation directory\n\n";
        }
        exit 1;
    }
# To find the standard supplied libraries in the local path
    use Cwd;
    $curDir = cwd;
}

use lib "$myLibPath";
use lib "$curDir";

use dKitCircuit;

#dKitCircuit->debug(1);

# This is a simulation to compare ADS vs. Hspice using the verification tool.
# It is a simple diode in parallel with a DC voltage source.
# The only variable swept is the forward diode voltage, Vd. The output is
# intended to be Capacitance and Current vs. Vd.

# NOTE: There is a required perl script, diode_create.pl, that must be
# executed before this script. This is so that a model with required
# parameters can be created for a diode. This script, diode_create.pl, only
# needs to be executed once. For more details, please see the contents of the
# diode_create.pl script.

# This test is written for the forward region (Vdc = 0 to 0.7) for the ndiode
# and pdiode diodes.

# Add instance of Diode and Voltage source.

$D1=dKitInstance->new('diode', "d1");
$D1->parameterValue('width', "300u");
$D1->parameterValue('length', "250u");
$D1->nodeName('n1', 1);
$D1->nodeName('n2', 0);

$vd=dKitInstance->new('V',"vd");
$vd->parameterValue('Vdc', .9);
$vd->nodeName('nplus', 1);
$vd->nodeName('nminus',0);

# Add DC Analysis Component
```

```
$DC1 = dKitAnalysis->new('DC',"DC1");
$DC1->parameterValue('device',"vd");
$DC1->parameterValue('parameter',"Vdc");

# Add Sweep Plan for Voltage sweep

$DCSweep = dKitAnalysis->new('SWEEPPLAN_LIN', "Plan1");
$DCSweep->parameterValue('start', 0);
$DCSweep->parameterValue('stop', .8);
$DCSweep->parameterValue('numPts', 16);
$DC1->addSweepPlan($DCSweep);

$SweepPlan1 = dKitAnalysis->new('SWEEPPLAN_PT', "Plan2");
$SweepPlan1->parameterValue('values', "-40 25 125");

$TempSweep = dKitAnalysis->new('SWEEP', "Sweep2");
$TempSweep->parameterValue('parameter', "temp");
$TempSweep->addSweepPlan($SweepPlan1);
$TempSweep->addSubAnalysis($DC1);

# Add output plans to measure parameters.

# $OUTPUTNODES = dKitAnalysis->new('OUTPUTPLAN_NODES', "out1");
# $OUTPUTNODES->parameterValue('nodes',1);
# $DC1->addOutputPlan($OUTPUTNODES);

$OUTPUTCURRENTS = dKitAnalysis->new('OUTPUTPLAN_CURRENTS', "out2");
$OUTPUTCURRENTS->parameterValue('deviceTerminals',"vd:1");
$DC1->addOutputPlan($OUTPUTCURRENTS);

$OUTPUTDOPS=dKitAnalysis->new('OUTPUTPLAN_DOPS', "out3");
$OUTPUTDOPS->parameterValue('deviceParameters', "vd:i d1:Cd");
$DC1->addOutputPlan($OUTPUTDOPS);


# Add Options Block

$OPTION1=dKitInstance->new('SIMULATOROPTION', "myOptions");
$OPTION1->parameterValue('spectreOptions', "lang=spectre reltol=1e-5 vabstol=1e-8 iabstol=1e-8
digits=10 save=lvlpub rawfmt=psfascii");
$OPTION1->parameterValue('adsOptions', "ResourceUsage=yes UseNutmegFormat=no ASCII_Rawfile=yes
I_AbsTol=1e-8 V_AbsTol=1e-8 I_RelTol=1e-5 V_RelTol=1e-5 Temp=27");
$OPTION1->parameterValue('hspiceOptions', "ingold=2 numdgt=10 nopage acct=0 dccap=1 abstol=1e-8
reltol=1e-5 absvdc=1e-8");


# Add Library Models
# The ads, hspice and spectre model files must be placed in the same directory as this script.

$MODLIB=dKitInstance->new('MODELLIBRARY', "my_model");
$MODLIB->parameterValue('adsModelLibrary', "my_model_nom_ads.net");
$MODLIB->parameterValue('hspiceModelLibrary', "my_model_nom_hspice.lib");
$MODLIB->parameterValue('spectreModelLibrary', "my_model_nom_spectre.scs");
$MODLIB->parameterValue('adsSectionName', "nominal");
$MODLIB->parameterValue('hspiceSectionName', "nominal");
$MODLIB->parameterValue('spectreSectionName', "nominal");

# Build the circuit

$testName = "test_diode_dc";  # Change this circuit name depending upon which
                              # model you are testing.
```

```
$CIRCUIT=dKitCircuit->new($testName);
$CIRCUIT->addSimulatorOption($OPTION1);
$CIRCUIT->addModelLibrary($MODLIB);
$CIRCUIT->addAnalysis($TempSweep);
$CIRCUIT->addInstance($D1);
$CIRCUIT->addInstance($vd);



@device = ("ndiode", "pdiode");

for $device (@device)
{
    $testName = "test_" . $device . "_dc";
    $CIRCUIT->circuitName($testName);
    $D1->parameterValue('model', $device);
    $CIRCUIT->simulate('ads','hspice','spectre');

    #### calculate statistics ...

    $adsDataP = dKitResults->getResults($CIRCUIT->circuitName, "ads");
    $spectreDataP = dKitResults->getResults($CIRCUIT->circuitName, "spectre");
    $hspiceDataP = dKitResults->getResults($CIRCUIT->circuitName, "hspice");

    $diff = dKitResults->compareDataRelative($adsDataP, $spectreDataP);
    $diff->writeCitifile("$CIRCUIT->circuitName" . "_asdiff.cti", "diffAdsSpectre");
    print $diff->calculateStatistics("$CIRCUIT->circuitName" . "_asstat.dat");

    $diff = dKitResults->compareDataRelative($adsDataP, $hspiceDataP);
    $diff->writeCitifile("$CIRCUIT->circuitName" . "_ahdiff.cti", "diffAdsHspice");
    print $diff->calculateStatistics("$CIRCUIT->circuitName" . "_ahstat.dat");

    $diff = dKitResults->compareDataRelative($hspiceDataP, $spectreDataP);
    $diff->writeCitifile("$CIRCUIT->circuitName" . "_hsdiff.cti", "diffHspiceSpectre");
    print $diff->calculateStatistics("$CIRCUIT->circuitName" . "_hsstat.dat");
}
```

diode_create.pl

The *diode_create.pl* script will create a two port network that will represent the diode used in the diode simulations. This script must be executed before any of the other diode scripts are ran. It is only required to be executed once. Once the script is executed, it will append *dKitTemplate.pm* to the template file. This is the required template for the diode.

```
#!/usr/bin/perl

BEGIN {
    if ($ENV{DKITVERIFICATION} eq "")
    {
        if ($ENV{HPEESOF_DIR} eq "")
        {
            print "\nPlease set environment variable DKITVERIFICATION\nto point to the design kit
verification installation directory\n\n";
            exit 1;
        } else
```

```
        {
            $ENV{DKITVERIFICATION} = "$ENV{HPEESOF_DIR}/design_kit/verification";
        }
    }
    $myLibPath = "$ENV{DKITVERIFICATION}/perl/lib";
    if ( ! -d $myLibPath)
    {
        if ($ENV{DKITVERIFICATION} eq "$ENV{HPEESOF_DIR}/design_kit/verification")
        {
            if ( ! -d "$ENV{HPEESOF_DIR}/design_kit/verification")
            {
                print "\nERROR: Unable to find verification module directory\nVerification tool not
installed at default\nlocation \$HPEESOF_DIR/design_kit/verification\n";
                print "Please set environment variable DKITVERIFICATION to point\nto the design kit
verification installation directory\n\n";
            } else
            {
                print "\nERROR: Unable to find verification module directory at\ndefault location
\$HPEESOF_DIR/design_kit/verification/perl/lib\n";
                print "Please set environment variable DKITVERIFICATION\nto point to the installation
directory\n\n";
            }
        } else
        {
            print "\nERROR : Unable to find verification module directory
\$DKITVERIFICATION/perl/lib\n\n";
            print "Please set environment variable DKITVERIFICATION\nto point to the design kit
verification installation directory\n\n";
        }
        exit 1;
    }
# To find the standard supplied libraries in the local path
    use Cwd;
    $curDir = cwd;
}

use lib "$myLibPath";
use lib "$curDir";


# This script will create a two port network that will represent the diode
# used in the diode simulations. This script must be executed before any of
# the other diode scripts are ran. It is only required to be executed once.
# Once executed, this script will append to the template file,
# dKitTemplate.pm, the required template for the diode.

### begin of script file

use dKitTemplate;


### add here the perl commands to create the new templates

$diode = dKitTemplate->new('diode');
$diode->requiredPrefix("d");
$diode->addNode("n1");
$diode->addNode("n2");
$diode->addParameter("model");
$diode->addParameter("length");
$diode->addParameter("width");
$diode->addParameter("temp");
```

```
$diode->netlistInstanceTemplate('ads', '<model>:#instanceName %n1 %n2 Length=<length> Width=<width>
[[temp=<temp>]]');
$diode->netlistInstanceTemplate('spectre', '#instanceName %n1 %n2 <model> length=<length>
width=<width> [[temp=<temp>]]');
$diode->netlistInstanceTemplate('hspice', '#instanceName %n1 %n2 <model> l=<length> w=<width>
[[temp=<temp>]]');

### update new dKitTemplate.pm file

#print dKitTemplate->componentTemplates2perl;

dKitTemplate->createTemplateModule;

### end of script module
```

diode_parameter.pl

The *diode_parameter.pl* script will add the Diode Parameter, *Cd*, to the Design Kit Parameter list for Hspice.

```
#!/usr/bin/perl

BEGIN {
    if ($ENV{DKITVERIFICATION} eq "")
    {
        if ($ENV{HPEESOF_DIR} eq "")
        {
            print "\nPlease set environment variable DKITVERIFICATION\nto point to the design kit
verification installation directory\n\n";
            exit 1;
        } else
        {
            $ENV{DKITVERIFICATION} = "$ENV{HPEESOF_DIR}/design_kit/verification";
        }
    }
    $myLibPath = "$ENV{DKITVERIFICATION}/perl/lib";
    if ( ! -d $myLibPath)
    {
        if ($ENV{DKITVERIFICATION} eq "$ENV{HPEESOF_DIR}/design_kit/verification")
        {
            if ( ! -d "$ENV{HPEESOF_DIR}/design_kit/verification")
            {
                print "\nERROR: Unable to find verification module directory\nVerification tool not
installed at default\nlocation \$HPEESOF_DIR/design_kit/verification\n";
                print "Please set environment variable DKITVERIFICATION to point\nto the design kit
verification installation directory\n\n";
            } else
            {
                print "\nERROR: Unable to find verification module directory at\ndefault location
\$HPEESOF_DIR/design_kit/verification/perl/lib\n";
                print "Please set environment variable DKITVERIFICATION\nto point to the installation
directory\n\n";
            }
        } else
        {
            print "\nERROR : Unable to find verification module directory
\$DKITVERIFICATION/perl/lib\n\n";
```

```
            print "Please set environment variable DKITVERIFICATION\nto point to the design kit
verification installation directory\n\n";
        }
        exit 1;
    }
# To find the standard supplied libraries in the local path
    use Cwd;
    $curDir = cwd;
}

use lib "$myLibPath";
use lib "$curDir";


use dKitParameter;

# This script will add the Diode Parameter, Cd, to the DK Parameter
# list for Hspice.

$Cd = dKitParameter->new('Cd');  # The diode diffusion capacitance.
$Cd->description("Diode Diffusion Capacitance");
$Cd->dialectReference('hspice', 'lx5ofdevice');
$Cd->dialectReference('ads', 'Cd');
$Cd->dialectReference('spectre', 'Cd');
$Cd->resultName("cd");

dKitParameter->createParameterModule;
```

Chapter 5: Comparing Simulation Results

This chapter describes the steps for comparing simulation results between the different simulators.

Viewing Your Results in a Data Display

Simulation results are stored in datasets (*.ds) and citifiles (*.cti). To view your results in an ADS Data Display:

1. Launch ADS and create a new project *<adsproject>* if necessary.
2. Copy all of your datasets to your ADS project's *data* directory (*<adsproject>*/data).
3. Launch the ADS Data Display server from your ADS window. From the ADS *Main* window, choose **Window > New Data Display.**
4. Create a template to view your results. For more information, refer to *"Using a Template in Your Display"* in the ADS *"Data Display"* documentation.
5. Select the dataset to view the results. If your parameters have been defined correctly, you should be able to use the *<dataset>.<simulator>.<parameter>* to access the different results for the different simulators, with *<simulator>* equal to ads_sim, spectre_sim or hspice_sim.

For more information on *datasets* (*.ds) and *citifiles* (*.cti), refer to Chapter 20 of the ADS *"Circuit Simulation"* manual.

Script Based Comparisons

The design kit model verification tool also contains some functionality to do script based comparisons of the simulation data. The following operations are needed:

- "Retrieving a Handle to the Results" on page 4-29
- "Comparing Two Simulator Results" on page 4-30
- "Performing a Simple Statistical Analysis" on page 4-31

If you want to do the comparison from within a different script other than the one used to run the simulation, the circuit debug level needs to be set to 1 (see debug facility of "Circuit Module (dKitCircuit.pm)" on page 4-36), in order not to delete the usable simulation results upon completion.

If you want to save the comparison results or view them using the ADS Data Display server, the functionality described in the following sections is also needed.

- "Writing the Results to Citifiles" on page 4-29
- "Converting Citifiles to Datasets" on page 4-30

For a description of these functions, refer to"Results Module (dKitResults.pm)" on page 4-41.

Example Script

This script is used to compare the simulation results between Spectre, Hspice and ADS for a circuit called *test_diode*.

The script starts as usual with the default header. When running this script, three sets of data will be created with the differences between the output data of two simulators: the point-to-point relative difference will be put in a citifile, and the statistics of these differences are saved in text files.

```
#!/usr/bin/perl

BEGIN {
    if ($ENV{DKITVERIFICATION} eq "")
    {
        if ($ENV{HPEESOF_DIR} eq "")
        {
            print "\nPlease set environment variable DKITVERIFICATION\nto point to the design kit
verification installation directory\n\n";
            exit 1;
        } else
        {
            $ENV{DKITVERIFICATION} = "$ENV{HPEESOF_DIR}/design_kit/verification";
        }
    }
    $myLibPath = "$ENV{DKITVERIFICATION}/perl/lib";
    if ( ! -d $myLibPath)
    {
        if ($ENV{DKITVERIFICATION} eq "$ENV{HPEESOF_DIR}/design_kit/verification")
        {
            if ( ! -d "$ENV{HPEESOF_DIR}/design_kit/verification")
            {
                print "\nERROR: Unable to find verification module directory\nVerification tool not
installed at default\nlocation \$HPEESOF_DIR/design_kit/verification\n";
                print "Please set environment variable DKITVERIFICATION to point\nto the design kit
verification installation directory\n\n";
            } else
            {
                print "\nERROR: Unable to find verification module directory at\ndefault location
\$HPEESOF_DIR/design_kit/verification/perl/lib\n";
                print "Please set environment variable DKITVERIFICATION\nto point to the installation
directory\n\n";
            }
        } else
        {
            print "\nERROR : Unable to find verification module directory
\$DKITVERIFICATION/perl/lib\n\n";
            print "Please set environment variable DKITVERIFICATION\nto point to the design kit
verification installation directory\n\n";
        }
        exit 1;
    }
# To find the standard supplied libraries in the local path
    use Cwd;
    $curDir = cwd;
}

use lib "$myLibPath";
use lib "$curDir";

use dKitResults;

$testName = "test_diode";

$adsDataP = dKitResults->getResults($testName, "ads");
$spectreDataP = dKitResults->getResults($testName, "spectre");
$hspiceDataP = dKitResults->getResults($testName, "hspice");

$diff = dKitResults->compareDataRelative($adsDataP, $spectreDataP);
$diff->writeCitifile("$testName" . "_asdiff.cti", "diffAdsSpectre");
print $diff->calculateStatistics("$testName" . "_asstat.dat");
```

```
$diff = dKitResults->compareDataRelative($adsDataP, $hspiceDataP);
$diff->writeCitifile("$testName" . "_ahdiff.cti", "diffAdsHspice");
print $diff->calculateStatistics("$testName" . "_ahstat.dat");

$diff = dKitResults->compareDataRelative($hspiceDataP, $spectreDataP);
$diff->writeCitifile("$testName" . "_hsdiff.cti", "diffHspiceSpectre");
print $diff->calculateStatistics("$testName" . "_hsstat.dat");
```

Chapter 6: Documenting Your Simulation Results

This chapter describes the fundamentals of using the ADS Electronic Notebook.

Creating Verification Documentation

The ADS Electronic Notebook enables you to create HTML documents using the designs and results within a project. The operations and features provided within the notebook enables you to:

- Automatically capture and update images from ADS schematics, layouts and data displays.
- Produce HTML formatted output of results which can be distributed easily without the necessity of ADS.
- Exercise the ability to import graphic images from external sources. This enables you to add documentation from other products.

To access the ADS Electronic Notebook Editor:

1. Choose **Tools > Electronic Notebook**. The ADS Electronic Notebook Editor appears.

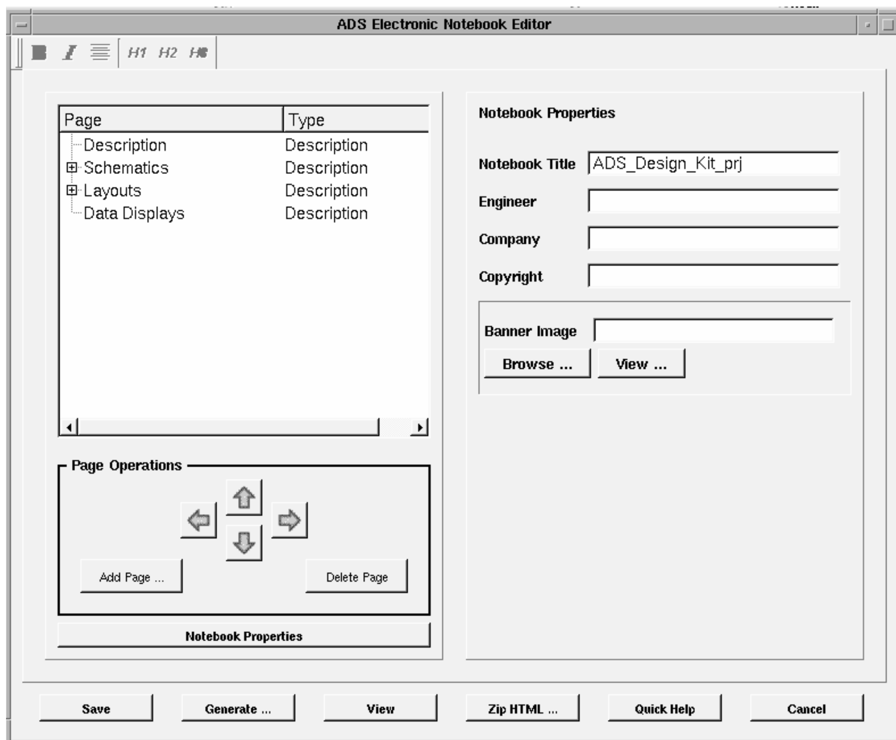


Figure 6-1. The ADS Electronic Notebook Editor

2. Enter the *Notebook Title*, *Engineer*, *Company* and *Copyright* information in the Notebook Properties section of the dialog.

Note You can return to and edit the Notebook Properties section of the dialog at any time by clicking the **Notebook Properties** button below the Page Operations section of the dialog.

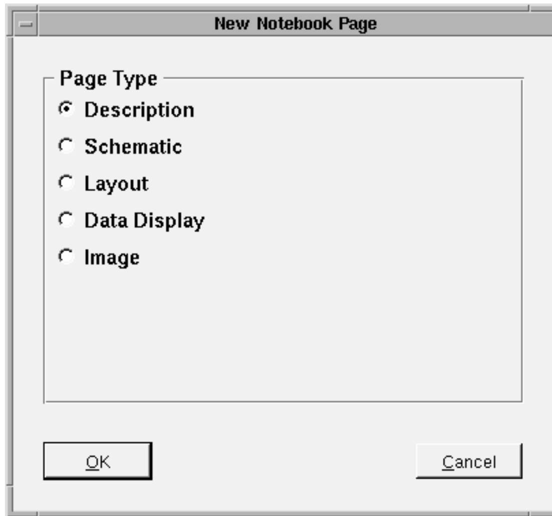
---

3. Use the **Browse** button to select an existing Banner Image to be displayed at the top of your document. Click the **View** button to preview the image.

Adding a Page

The Electronic Notebook supports a variety of page types. Generally, a page consists of a beginning text area, an image and a bottom text area. The text area is HTML formatted. This means that the text in this area may make use of any of the standard HTML formatting commands. To add a new page:

1. Click the **Add Page** button. The New Notebook Page dialog box appears.



2. Select the appropriate Page Type from the list of options. For more information, refer to "Page Types" on page 6-4.
3. Click **OK** to continue or **Cancel** to abort the operation. If you clicked **OK**, the page is inserted below the currently selected page. It is not inserted into the selected page's folder. If this is desired, use the right arrow operation after the page is inserted.

Page Types

- **Folders** There are no explicit folder pages in the notebook. Any page can become a folder. This is accomplished by moving a page beneath the desired folder page and moving it to the right by using the right arrow button. Similarly, moving a page to the left takes it out of its folder.
- **Description** This places a blank page into the notebook. This page has only one text area. It is recommended that general descriptions of topic areas be placed on these pages.
- **Schematic** This places a schematic image on the page and has the standard two text areas.
- **Layout** This places a layout image on the page and has the standard two text areas.
- **Data Display** This places a data display image on the page and has the standard two text areas.
- **Image** This places an external image on the page and has the standard two text areas. The image can be a jpeg, gif, bmp or xwd image. The image is automatically converted to jpeg format during page generation so that complete portability is possible.

Notebook Operations

- **Save** Saves the current notebook data. The HTML data is not created when this data is saved.
- **Generate** Saves the current notebook data. The HTML data is then created and a web browser is opened up to view the resulting notebook.
- **View** Opens the current HTML notebook in a browser window. This command does not save the current configuration or generate new HTML data.
- **Zip HTML** Saves the current notebook HTML data in a zip file. A prompt appears requesting the location of the zip file.
- **Quick Help** Displays the on line help.
- **Cancel** Exits the ADS Electronic Notebook.

Page Operations

Page operations affect the order of the pages within the notebook. The notebook is based on a hierarchical structure. Each page can become a folder simply by having another page underneath it.

Page Movement

The following operations move the pages around within the notebook.

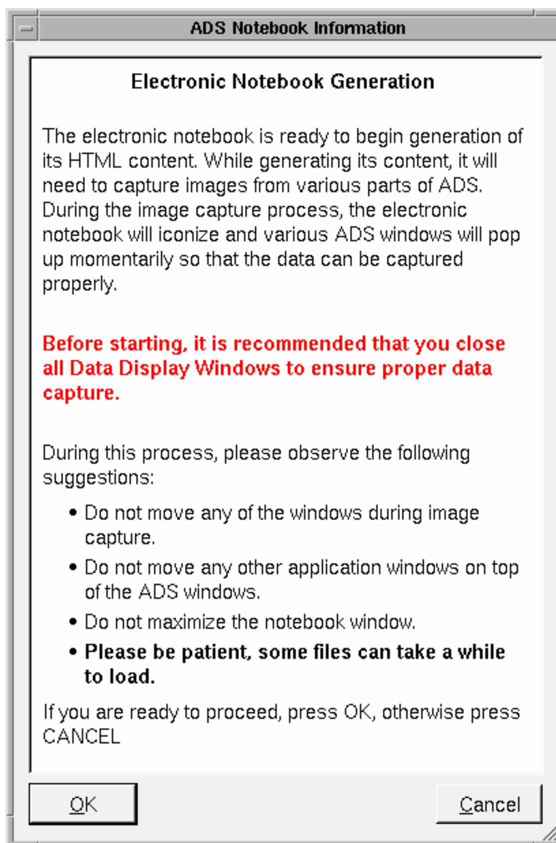
- **Up Arrow** Moves the selected page up. If the selected page is at the top of a list, it will be unable to move.
- **Down Arrow** Moves the selected page down. If the selected page is at the bottom of a list, it will be unable to move.
- **Left Arrow** Moves the selected page left. This will move the page out of the current list it's in and into the one above it. If the page is at the top level, it will be unable to move.
- **Right Arrow** Moves the selected page right. This moves the current page into the hierarchy of the page above it. If the page is already within the hierarchy of the preceding page, it will be unable to move.

Page Creation and Deletion

- **Add Page** This operation brings up a dialog which allows the addition of any of the default page types. The page is inserted below the currently selected page. It is not inserted into the selected page's folder. If this is desired, use the right arrow operation after the page is inserted.
- **Delete Page** This operation brings up a dialog confirming the deletion of the selected page. If a page has pages underneath it, a special dialog will confirm the deletion of the page and all its children.

Notebook Generation

Notebook generation consists of two parts, the HTML file creation and the image capture. The HTML generation for the notebook is done automatically whenever the notebook is generated using the Generate command.



The Save command does not generate the HTML, it only saves the notebook data, so that when the notebook is restarted it can be resumed from its last state.

The images which are used in the notebook are image captures taken directly from the screen. During this process, the notebook is iconized so that it will not get in the way of the image capture process. After all the images have been captured, it is maximized back to its last size. It is important to remember that this process is a direct screen capture and any windows placed on top of an ADS window during this process will be captured instead of the desired ADS image.

During the image capture process, each desired design or data display is opened and then resized appropriately for correct screen capture. Again, these designs do not

open any faster than they normally would, so patience is required for large schematics and data display files.

During the image capture process, only those designs and data displays which have changed since the last image capture are captured. However, if the image should not be updated automatically, the check box on the desired page, **Update Images Automatically**, should be unchecked. If this box is unchecked, the image will not be recaptured when the design or data is updated.

Appendix A: Template List

This appendix contains a list of default templates currently available in the standard list of templates.

The following sections contain the currently defined default templates:

- "Analysis Templates" on page A-2
- "Outputplan Templates" on page A-4
- "Sweepplan Templates" on page A-6
- "Simulator Options Templates" on page A-8
- "ModelLibrary Template" on page A-9
- "Components/Instance Templates" on page A-10
- "Circuit Initialization" on page A-24

Note In each of the tables listed in this appendix, the parenthesis containing the specific simulators (*<simulator>*) indicates that the parameter is available in that particular simulator.

Analysis Templates

This section contains the default analysis templates.

For all of these templates the parameter defines the parameter to be swept, if that parameter is part of a device, device should be set to the name of the instance defining the device. For more information on adding a sweep plan and output plans, refer to "Adding a Sweepplan" on page 4-23 and "Adding an OutputPlan" on page 4-24.

AC analysis

Table A-1. AC Analysis Template

TemplateName:	AC		
Description:	Small signal AC analysis		
Parameters:			
device	(ads)	(hspice)	(spectre)
frequency	(ads)	(hspice)	(spectre)
parameter	(ads)	(hspice)	(spectre)

DC analysis

Table A-2. DC Analysis Template

TemplateName:	DC		
Description:	DC analysis		
Parameters:			
device	(ads)	(hspice)	(spectre)
parameter	(ads)	(hspice)	(spectre)

Noise analysis

Table A-3. Noise Analysis Template

TemplateName:	NOISE		
Description:	Noise analysis		
Parameters:			
device	(ads)	(hspice)	(spectre)
frequency	(ads)	(hspice)	(spectre)
noiseNode	(ads)	(hspice)	(spectre)
noiseReference		(hspice)	
parameter	(ads)	(hspice)	(spectre)

S-parameter analysis

Table A-4. S-parameter Analysis Template

TemplateName:	SP		
Description:	S-parameter analysis		
Parameter:			
device	(ads)	(hspice)	(spectre)
frequency	(ads)	(hspice)	(spectre)
parameter	(ads)	(hspice)	(spectre)

SWEEP analysis

Table A-5. SWEEP Analysis Template

TemplateName:	SWEEP		
Description:	Sweep analysis		
Parameters:			
device	(ads)	(hspice)	(spectre)
parameter	(ads)	(hspice)	(spectre)

Outputplan Templates

This section contains the default outputplan templates.

ADS does not support OUTPUTPLAN_CURRENTS, and OUTPUTPLAN_DOPS will give all parameters for all devices if at least one parameter of one device is given. Outputplans added to S-parameter analyses will be ignored.

Outputplan for Device Currents

Table A-6. Outputplan for Device Currents Template

TemplateName:	OUTPUTPLAN_CURRENTS		
Description:	Default outputplan for device currents.		
Format:	List of device:terminal, supplied as a space delimited string.		
Parameters:			
deviceTerminals		(hspice)	(spectre)

Outputplan for Device Operating Point (DOP)

Table A-7. Current Outputplan Template

TemplateName:	OUTPUTPLAN_DOPS		
Description:	Default outputplan for device operating points.		
Format:	List of device:parameter, supplied as a space delimited string.		
Parameters:			
deviceParameters	(ads)	(hspice)	(spectre)

Outputplan for Nodes

Table A-8. Current Outputplan Template

TemplateName:	OUTPUTPLAN_NODES
Description:	Default outputplan for nodes.

Table A-8. Current Outputplan Template

Format:	List of Nodenames, supplied as a space delimited string.		
Parameters:			
nodes	(ads)	(hspice)	(spectre)

Sweepplan Templates

This section contains the default sweepplan templates.

It is possible to add an offset or scale the values of the sweeplan for a specific simulator using the *<dialect>*Scale and *<dialect>*Offset parameters. To re-normalize the values in the results, the functions `dKitResults->sweepVaribleScale` and `dKitResults->sweepVariableOffset` can be used.

Linear Sweep Plan

Table A-9. Linear Sweep Plan Template

TemplateName:	SWEEPPLAN_LIN		
Description:	Linear sweep plan.		
Parameters:			
numPts	(ads)	(hspice)	(spectre)
start	(ads)	(hspice)	(spectre)
step	(ads)	(hspice)	(spectre)
stop	(ads)	(hspice)	(spectre)
hspiceScale		(hspice)	
hspiceOffset		(hspice)	
adsScale	(ads)		
adsOffset	(ads)		
spectreScale			(spectre)
spectreOffset			(spectre)

Discrete Point Sweepplan

The value parameter is a list of values, supplied as a space delimited string. For example:

```
$PointSweep->parameterValue('values', "1 2 3 4");
```

Table A-10. Discrete Point Sweep Plan Template

TemplateName:	SWEEPPLAN_PT		
Description:	Sweep plan consisting of discrete points.		
Parameters:			
values	(ads)	(hspice)	(spectre)
hspiceScale		(hspice)	
hspiceOffset		(hspice)	
adsScale	(ads)		
adsOffset	(ads)		
spectreScale			(spectre)
spectreOffset			(spectre)

Simulator Options Templates

This section contains the default simulator options templates.

Circuit Simulator Options

For each simulator, the options should be specified as a list using a space delimited string as they need to be specified for the simulator. For example:

```
$myOption->parameterValue('hspiceOptions', "nopage acct=0 dccap=1
numdgt=10");
```

Table A-11. Circuit Simulator Options Template

TemplateName:	SIMULATOROPTION		
Description:	Circuit options.		
Parameters:			
adsOptions	(ads)		
hspiceOptions		(hspice)	
spectreOption			(spectre)

Circuit Temperature

Table A-12. Circuit temperature Template

TemplateName:	TEMPERATURE		
Description:	Circuit Temperature.		
Parameters:			
temperature	(ads)	(hspice)	(spectre)

ModelLibrary Template

This section contains the default model library templates.

Model Library

*<dialect>*ModelLibrary should contain the path to the *<dialect>*library, *<dialect>*SectionName is the section of that library to be activated, if it is omitted the whole library is included.

Table A-13. Model Library Template

TemplateName:	MODELLIBRARY		
Description:	Includes one modellibrary.		
Parameters:			
adsModelLibrary	(ads)		
adsSectionName	(ads)		
hspiceModelLibrary		(hspice)	
hspiceSectionName		(hspice)	
spectreModelLibrary			(spectre)
spectreSectionName			(spectre)

Components/Instance Templates

This section contains a description of the default instance templates to be added to the circuit using the addInstance function from the Circuit module.

Parameter

This template defines a circuit parameter. `value` can be a numerical value (single parameter) or an equation. Be sure to add all single parameters to the circuit before adding equation parameters, otherwise the simulation results can be very awkward.

Table A-14. Parameter Template

TemplateName:	PARAMETER		
Description:	Netlist parameter.		
Parameters:			
value	(ads)	(hspice)	(spectre)

Capacitor

Table A-15. Capacitor Template

TemplateName:	C		
RequiredPrefix:	c		
Description:	Capacitor.		
Nodes:	n1, n2		
Parameters:			
capacitance	(ads)	(hspice)	(spectre)
dtemp		(hspice)	(spectre)
initialCurrent	(ads)	(hspice)	(spectre)
multiplicity	(ads)	(hspice)	(spectre)
scale		(hspice)	
temp	(ads)		
tempCoef1	(ads)	(hspice)	(spectre)

Table A-15. Capacitor Template

tempCoef2	(ads)	(hspice)	(spectre)
tnom	(ads)		

Capacitor with Model

Table A-16. Capacitor with Model Template

TemplateName:	CM		
RequiredPrefix:	c		
Description:	Capacitor with model.		
Nodes:	n1, n2		
Parameters:			
capacitance	(ads)	(hspice)	(spectre)
dtemp		(hspice)	(spectre)
initialCurrent	(ads)	(hspice)	(spectre)
length	(ads)	(hspice)	(spectre)
modelName	(ads)	(hspice)	(spectre)
multiplicity	(ads)	(hspice)	(spectre)
scale		(hspice)	
temp	(ads)		
tempCoef1	(ads)	(hspice)	(spectre)
tempCoef2	(ads)	(hspice)	(spectre)
tnom	(ads)		
width	(ads)	(hspice)	(spectre)

Diode

Table A-17. Diode Template

TemplateName:	D		
RequiredPrefix:	d		
Description:	diode.		
Nodes:	nplus, nminus		
Parameters:			
area	(ads)	(hspice)	(spectre)
dtemp		(hspice)	
initialVoltage		(hspice)	
lengthPolySilicon		(hspice)	
length		(hspice)	
lengthMetalCapacitor		(hspice)	
modelName	(ads)	(hspice)	(spectre)
multiplicity	(ads)	(hspice)	(spectre)
off		(hspice)	
periphery	(ads)		
peripheryJunction		(hspice)	
temp	(ads)		
tnom	(ads)		
width		(hspice)	
widthMetalCapacitor		(hspice)	
widthPolySilicon		(hspice)	

Independent Current Source

Table A-18. Independent Current Source Template

TemplateName:	I		
RequiredPrefix:	i		
Description:	Independent current source.		
Nodes:	nplus, nminus		
Parameters:			
Iac_Mag	(ads)	(hspice)	(spectre)
Iac_Phase=0	(ads)	(hspice)	(spectre)
Idc	(ads)	(hspice)	(spectre)
multiplicity		(hspice)	(spectre)
transientFunction	(ads)	(hspice)	
type			(spectre)
typeParams			(spectre)

JFET

Table A-19. JFET Template

TemplateName:	J		
RequiredPrefix:	j		
Description:	JFET.		
Nodes:	nd, ng, ns, nb		
Parameters:			
area	(ads)	(hspice)	(spectre)
dtemp		(hspice)	
length		(hspice)	
modelName	(ads)	(hspice)	(spectre)
multiplicity	(ads)	(hspice)	(spectre)
off		(hspice)	
region	(ads)		(spectre)
temp	(ads)		
tnom	(ads)		
vdsval		(hspice)	
vgsval		(hspice)	
width		(hspice)	

Mutual Inductor

Table A-20. Mutual Inductor Template

TemplateName:	K		
RequiredPrefix:	k		
Description:	Mutual Inductor.		
Nodes:			
Parameters:			
coupling	(ads)	(hspice)	(spectre)
inductor1	(ads)	(hspice)	(spectre)
inductor2	(ads)	(hspice)	(spectre)
mutualInductance	(ads)		

Inductor

Table A-21. Inductor Template

TemplateName:	L		
RequiredPrefix:	l		
Description:	Inductor.		
Nodes:	n1, n2		
Parameters:			
dtemp		(hspice)	
inductance	(ads)	(hspice)	(spectre)
initialCurrent	(ads)	(hspice)	(spectre)
multiplicity	(ads)	(hspice)	(spectre)
resistance	(ads)	(hspice)	(spectre)
scale		(hspice)	
temp	(ads)		(spectre)
tempCoef1	(ads)	(hspice)	(spectre)
tempCoef2	(ads)	(hspice)	(spectre)
tnom	(ads)		(spectre)

Inductor with Model

Table A-22. Inductor with Model Template

TemplateName:	LM		
RequiredPrefix:	l		
Description:	Inductor with Model.		
Nodes:	n1, n2		
Parameters:			
dtemp			(spectre)
inductance	(ads)		(spectre)
initialCurrent	(ads)		(spectre)
modelName	(ads)		(spectre)
multiplicity	(ads)		(spectre)
resistance	(ads)		
temp	(ads)		
tempCoef1	(ads)		
tempCoef2	(ads)		
tnom	(ads)		

MOSFET

Table A-23. MOSFET Template

TemplateName:	M		
RequiredPrefix:	m		
Description:	MOSFET.		
Nodes:	nd, ng, ns, nb		
Parameters:			
drainDiffusionArea	(ads)	(hspice)	(spectre)
drainDiffusionSquares	(ads)	(hspice)	(spectre)
drainPerimeter	(ads)	(hspice)	(spectre)
drainResistanceAddl		(hspice)	
dtemp		(hspice)	(spectre)
geoSharing		(hspice)	
length	(ads)	(hspice)	(spectre)
modelName	(ads)	(hspice)	(spectre)
multiplicity	(ads)	(hspice)	(spectre)
off		(hspice)	
region	(ads)		(spectre)
sourceDiffusionArea	(ads)	(hspice)	(spectre)
sourceDiffusionSquares	(ads)	(hspice)	(spectre)
sourcePerimeter	(ads)	(hspice)	(spectre)
sourceResistanceAddl		(hspice)	
temp	(ads)		
thresholdVoltageShift		(hspice)	
tnom	(ads)		
vbsval		(hspice)	
vdsval		(hspice)	
vgsval		(hspice)	
width	(ads)	(hspice)	(spectre)

PORT

Table A-24. Port Template

TemplateName:	PORT		
RequiredPrefix:	v		
Description:	Port for S-Parameter Analysis.		
Nodes:	nplus, nminus		
Parameters:			
Vac_Mag		(hspice)	
Vdc	(ads)	(hspice)	(spectre)
portNr	(ads)		(spectre)
referenceImpedance	(ads)		(spectre)
transientFunction		(hspice)	

BJT

Table A-25. BJT Template

TemplateName:	Q		
RequiredPrefix:	q		
Description:	Bipolar Junction Transistor.		
Nodes:	nc, nb, ne, ns		
Parameters:			
areaBasis		(hspice)	
areaCollector		(hspice)	
areaEmittor	(ads)	(hspice)	(spectre)
dtemp		(hspice)	(spectre)
modelName	(ads)	(hspice)	(spectre)
multiplicity	(ads)	(hspice)	(spectre)
off		(hspice)	
region	(ads)		(spectre)
temp	(ads)		
tnom	(ads)		
vbeval		(hspice)	
vceval		(hspice)	

Resistor with Model

Table A-26. Resistor with Model Template

TemplateName:	RM		
RequiredPrefix:	r		
Description:	Resistor with model.		
Nodes:	n1, n2		
Parameters:			
dtemp			(spectre)
length	(ads)		(spectre)
modelName	(ads)		(spectre)
multiplicity	(ads)		(spectre)
narrow	(ads)		
narrowLength	(ads)		
narrowWidth	(ads)		
noise			(spectre)
resistance	(ads)		(spectre)
sheetResistance	(ads)		
temp	(ads)		
tempCoef1	(ads)		(spectre)
tempCoef2	(ads)		(spectre)
tnom	(ads)		
width	(ads)		(spectre)

Independent Voltage Source

Table A-27. Independent Voltage Source Template

TemplateName:	V		
RequiredPrefix:	v		
Description:	Independent voltage source.		
Nodes:	nplus, nminus		
Parameters:			
Vac_Mag	(ads)	(hspice)	(spectre)
Vac_Phase=0	(ads)	(hspice)	(spectre)
Vdc	(ads)	(hspice)	(spectre)
multiplicity			(spectre)
transientFunction	(ads)	(hspice)	
type			(spectre)
typeParams			(spectre)

MESFET

Table A-28. MESFET Template

TemplateName:	Z		
RequiredPrefix:	z		
Description:	MESFET.		
Nodes:	nd, ng, ns, nb		
Parameters:			
area	(ads)	(hspice)	(spectre)
dtemp		(hspice)	
length		(hspice)	
modelName	(ads)	(hspice)	(spectre)
multiplicity		(hspice)	(spectre)
off		(hspice)	
region	(ads)		(spectre)
vdsval		(hspice)	
vgsval		(hspice)	
width		(hspice)	

Circuit Initialization

This section contains the default circuit initialization templates.

Circuit End Deck Card

Table A-29. Circuit End Deck Card Template

TemplateName:	CIRCUITENDDECKCARD		
Description:	If defined, added as last card of the deck.		
Parameters:			
adsEndDeckcard	(ads)		
hspiceEndDeckcard=.end		(hspice)	
spectreEndDeckcard			(spectre)

Command to Invoke Simulator

Table A-30. Invoke Simulator Command Template

TemplateName:	CIRCUITINVOKECOMMAND		
Description:	Command to invoke the simulator.		
Parameters:			
cleanup	(ads)	(hspice)	(spectre)
netlistFilename	(ads)	(hspice)	(spectre)

Netlistnames for the Different Simulators

Table A-31. Netlistnames for Different Simulators Template

TemplateName:	CIRCUITNETLISTNAME		
Description:	Netlist names for the different simulators.		
Parameters:			
adsNetlistSuffix=.ckt	(ads)		
hspiceNetlistSuffix=.hsp		(hspice)	
projectName			(spectre)
spectreNetlistSuffix=.scs	(ads)	(hspice)	(spectre)

Circuit Start Deck Card

Table A-32. Circuit Start Deck Card Template

TemplateName:	CIRCUITSTARTDECKCARD		
Description:	If defined, added as first card of the deck.		
Parameters:			
adsStartDeckcard	(ads)		
hspiceStartDeckcard		(hspice)	
spectreStartDeckcard= simulator lang=spectre			(spectre)

Index

A
AC analysis, 3-15, 3-16
ActiveState, 2-3
ADS, 1-1, 3-5, 3-17, 3-18, 5-2
    circuit simulator, 4-19
    Data Display, 1-2, 1-4, 3-11, 3-23, 4-30, 4-31, 5-2
    Electronic Notebook, 1-2, 1-4
    installation directory, 2-4
    model library file, 3-14
Analyses, 4-21
analyses, 3-15
    adding, 4-17
    cascaded, 3-16
    outputplans, 4-24
    removing, 4-17
    retrieving analysisNames, 4-23
    sweepplans, 4-23
analysis, 3-1, 3-4
    AC, 3-15, 3-16
    DC, 3-15, 3-16
    handles, 4-22
    NOISE, 3-15, 3-17
    SP, 3-15, 3-17
    standard analyses, 3-4
arguments, 2-3
B
basenames, 4-19
bash shell, 2-3
C
C shell, 2-4
case sensitive, 3-12
circuit options, 3-12
CIRCUITENDDECKCARD, 4-18
CIRCUITNETLISTNAME, 4-19
CIRCUITSTARTDECKCARD, 4-18
CITI files, 3-11, 3-23, 4-17, 4-29, 4-30, 5-2
    converting to dataset, 4-30
    creating, 4-29
    merging, 4-29, 4-31
commands
    dKitSetupWork.pl, 2-6
    dKitVrun, 2-4, 2-5
    mount, 2-5
comment character (#), 2-1
components
    adding, 4-16
    component definition, 3-14
    connecting, 3-15
    custom, 3-5
    standard, 3-4
compound statements, 2-2
convergence tolerances, 3-12
corner analysis, 3-13
currents, 3-17
custom templates, 1-2
customization files, 3-6
cygnus, 2-3
    shell, 2-5, 3-1
D
dataset, 3-11, 3-22, 3-23, 4-5, 4-17, 4-19, 4-30, 4-31
DC analysis, 3-15, 3-16
declarations, 2-1
default outputplan, 3-17
design kit
    modules, 3-2
device operating points, 3-17
diode_create.pl, 4-46
diode_parameter.pl, 4-48
diode_test_dc.pl, 4-43
directories
    bin, 2-3, 2-4
    custom, 2-5
    data, 3-23
    install, 2-4
    perl, 3-4
dKitSetupWork.pl, 3-22
DKITVERIFICATION, 2-4, 2-5
dKitVrun, 2-4, 2-5
documentation
    retrieving, 4-4
documenting
    parameters, 4-11
    templates, 4-4
E
endDackCard, 4-18
engineering notation, 4-22
environment variables, 2-4
    DKITVERIFICATION, 2-5
equations, 3-13, 4-16
errors, 4-6

can’t open perl script, 2-4
error log, 1-2
out of memory error, 4-19
template already defined, 3-5, 4-8
expressions, 2-2, 4-6
F
files
.ael, 3-6
.ckt, 4-19
.cshrc, 2-4
.cti, 3-11, 3-23, 4-17, 4-29, 4-30, 5-2
.ds, 3-11, 3-22, 3-23, 4-17
.hsp, 4-19
.pl, 2-3
.pm, 4-1
.profile, 2-4
.scs, 4-19
dKitAnalysis.pm, 4-39
dKitCircuit.pm, 4-7, 4-14, 4-36
dKitInstance.pm, 4-21, 4-38
dKitParameter.pm, 3-22, 4-9, 4-13, 4-35
dKitParameterCreate.pl, 3-8
dKitResults.pm, 4-27, 4-41
dKitSetupWork.pl, 3-22
dKitTemplate.pm, 3-4, 3-7, 3-22, 4-3, 4-8, 4-33
dKitTemplateCreate.pl, 3-4, 4-5, 4-7
dKitTemplateList.pl, 3-4
header.pl, 3-4, 3-5, 3-8, 3-11
intermediate, 4-15
model files, 3-13
model library files, 3-13
results file, 3-8
FOOTERTEMPLATE, 4-8
functions, 1-4, 2-2, 4-1
addAnalysis(), 3-18, 4-17, 4-36
addInstance(), 4-16, 4-36
addModelLibary(), 4-16, 4-36
addNode(), 4-7, 4-33
addOutputPlan(), 4-24, 4-39
addParameter(), 4-7, 4-18, 4-33, 4-36
addSimulatorOption(), 4-15, 4-36
addSubAnalysis(), 3-16, 4-24, 4-39
addSweepPlan(), 4-23, 4-39
analysisName, 4-39
calculateStatistics(), 4-31, 4-41
circuitName(), 4-15, 4-17, 4-36
compareDataAbsolute(), 4-30, 4-41
compareDataRelative(), 4-30, 4-41
convertCitifile2Dataset(), 4-30, 4-41
createParameterModule, 4-35
createSubcircuit(), 4-7, 4-33
createTemplateModule, 4-33
debug(), 4-1, 4-18, 4-19, 4-33, 4-35, 4-36, 4-38, 4-39, 5-1
deleteResultsFromMemory(), 4-19, 4-36
description(), 4-4, 4-11, 4-35
dialectReference(), 4-11, 4-35
dKitCircuit, 4-36
dumpDKitXrefs, 4-35
dumpTemplates, 4-34
freeMemory(), 4-31, 4-41
getResults(), 4-29, 4-41
getTemplate(), 4-4, 4-34
instanceName, 4-38
listMissingParameterDefinitions, 4-35
mergeCitifiles(), 4-31, 4-41
netlist(), 4-18, 4-36
netlistInstanceTemplate(), 4-5, 4-34
new(), 3-1, 3-5, 3-8, 3-11, 3-12, 4-4, 4-11, 4-15, 4-22, 4-24, 4-34, 4-35, 4-37, 4-38, 4-39
nodeName(), 3-15, 4-23, 4-38
parameterValue(), 4-7, 4-18, 4-22, 4-34, 4-37, 4-38, 4-39
readData(), 4-29, 4-41
removeAnalysis(), 4-17, 4-37
removeInstance(), 4-17, 4-37
removeModelLibrary(), 4-16, 4-37
removeOutputPlan(), 4-24, 4-39
removeSimulatorOption(), 4-15, 4-37
removeSubAnalysis(), 4-25, 4-39
removeSweepPlan(), 4-24, 4-40
requiredPrefix(), 4-4, 4-34
resultName(), 4-12, 4-35
simulate(), 4-17, 4-19, 4-37
sweepVariableOffset(), 4-28, 4-42
sweepVariableScale(), 4-28, 4-42
template, 4-38, 4-40
templateName, 4-34
templates2perl, 4-34
writeCitifile(), 4-29, 4-42
G
global noise temperature, 3-12
H
handles, 3-1, 3-4, 3-5, 3-6, 3-11, 3-12, 3-13, 3-14, 3-16, 4-4
analysis, 4-22

circuit, 4-15
item, 4-22
lost, 4-29
parameter, 4-11
hash tables, 4-8, 4-13
%subcircuitVocabularyHash, 4-8
HEADERTEMPLATE, 4-8
hpeesofsim, 2-1, 2-6
Hspice, 1-1, 2-1, 2-6, 3-12, 3-13, 3-16, 3-18, 4-4, 4-12, 4-16, 4-18, 4-19, 5-2
I
instance, 3-1
Instances, 4-21
instances, 3-14
adding, 4-16
removing, 4-17
retrieving instanceNames, 4-23
INSTANCETEMPLATE, 4-8
item definition, 3-6
item names, 3-12
items, 3-1, 3-4
K
Korn shell, 2-4
L
layout, 3-15
library, 3-13
files, 3-6
Section designator, 3-13, 3-14
verification, 1-1
M
maximum, 4-31
mean, 4-31
minimum, 4-31
model files, 1-1, 3-13
model verification, 1-2, 1-3
modelLibrary, 3-13
adding a path, 4-16
removing a path, 4-16
modifiers, 2-2
modules, 4-1
analysis, 4-39
circuit, 4-5, 4-7, 4-14, 4-36
instance, 4-21, 4-38
parameter, 1-2, 3-22, 4-9, 4-13, 4-35
perl modules, 4-1, 4-32
results, 4-27, 4-28, 4-41
template, 1-2, 2-6, 3-4, 3-7, 3-22, 4-3, 4-8, 4-33
N
netlist, 3-1, 3-4, 4-3
format, 3-14, 4-5
function, 4-5
parameters
adding, 4-16
suffix, 4-19
template, 4-18
nodes, 3-6, 3-15, 3-17, 4-5, 4-6, 4-7
node names, 3-15, 4-23
NOISE analysis, 3-15, 3-17
O
offset value, 4-28
operators, 2-2
options, 3-12
removing, 4-15
simulator, 4-15
OUTPUTPLAN, 3-15, 3-16
adding, 4-24
OUTPUTPLAN_CURRENTS, 3-17
OUTPUTPLAN_DOPS, 3-17
OUTPUTPLAN_NODES, 3-17
removing, 4-24
P
parameters, 3-1, 3-6, 3-8
adding template parameters, 4-7
circuit, 3-4, 4-18
custom, 3-8
default, 2-6, 3-8
template parameter, 4-7
definitions, 3-8
dialect names, 4-11
displaying, 4-12
documenting, 4-11
global parameters, 3-16
missing, 4-13
module, 1-2
name, 4-6
not translated, 3-22, 4-35
Offset, 4-25
optional, 4-6
retrieving values, 4-22
Scale, 4-25
scripts, 1-2
Section parameter, 3-13
simulator specific parameter, 3-22
storing definitions, 4-13
swept, 3-16, 4-25

values, 3-14
PATH, 2-4, 2-6, 3-13
Perl, 1-1, 1-2, 2-1
ActivePerl, 2-2
eval function, 4-6
executable, 2-3
interpreter, 2-2, 2-3, 3-4
modules, 4-32
script header, 1-2
pins, 3-15, 4-6, 4-7
platforms
PC, 2-3, 2-5, 3-1
unix, 2-2, 2-3, 2-4, 3-1
prefixs
required, 4-4
R
relative paths, 3-13
results file, 3-8
S
scale value, 4-28
script based comparisons, 5-1
scripts
advanced, 4-1
basic, 3-1
diode_create.pl, 4-46
diode_parameter.pl, 4-48
diode_test_dc.pl, 4-43
dKitAnalysis.pm, 4-39
dKitCircuit.pm, 4-7, 4-14, 4-36
dKitInstance.pm, 4-21, 4-38
dKitParameter.pm, 3-22, 4-9, 4-13, 4-35
dKitParameterCreate.pl, 3-8
dKitResults.pm, 4-27, 4-41
dKitSetupWork.pl, 3-22
dKitTemplate.pm, 3-4, 3-7, 3-22, 4-3, 4-8, 4-33
dKitTemplateCreate.pl, 3-4, 4-5, 4-7
dKitTemplateList.pl, 3-4
header.pl, 3-5, 3-8, 3-11
parameter, 1-2
perl, 2-1, 3-1
script headers, 3-1, 3-2
test, 1-2
setup, 1-4, 2-1, 3-1
simulation, 1-1, 3-1
circuit, 4-17
results, 1-2, 4-19, 4-30, 4-31, 5-1
reading, 4-29
running, 4-19
supported simulators, 2-1
simulator, 3-4
definition, 3-18
dialects, 4-3, 4-5, 4-17
name, 4-5
netlists, 4-15
options, 3-12, 4-15
parameter values, 3-12
source code
modifying, 2-5, 4-6
S-parameter analysis, 3-15, 3-17
Spectre, 1-1, 2-1, 2-6, 3-17, 3-18, 4-18, 4-19, 4-25, 5-2
startDeckCard, 4-18
statements, 2-1
#define, 3-14
#function, 4-5
#instanceName, 3-6, 4-5
statistical analysis, 4-31
strings
quotes, 2-2, 3-1, 4-2
subanalysis
adding, 4-24
removing, 4-25
subinstances, 4-21
SWEEP, 3-15, 4-11
SWEEPPLAN, 3-15, 4-25
adding, 4-23
removing, 4-24
symbolic links, 2-3
symbols
#, 2-1
#!, 2-1, 2-3
T
temperature, 3-12, 3-16
templates, 3-4, 3-8
custom, 1-2, 3-5
data display, 3-23
dialects, 3-6
displaying, 4-8
standard, 3-4
sweepplan, 4-25
template definition, 3-6, 4-8, 4-22
template module, 1-2, 2-6
terminals, 3-15
test
circuit, 3-1, 3-4, 3-5, 3-8, 3-10, 3-18, 3-19
scripts, 1-2, 5-2
traces, 3-15

translated models, 1-1
translation rules, 3-4
    parsing, 4-7
tutorial example, 3-1
U
user interface, 2-1
V
variables, 3-13, 4-6, 4-18
    dependent, 4-31
    mapping, 4-9
    sweepvariable, 4-26
verification tool
    requirements, 3-1
voltage sources, 3-17
W
warnings, 3-12, 3-22
Windows, 2-2
    WinNT, 2-3