Agilent Technologies

Advanced Design System 2002

Design Kit Development

February 2002

Notice

Warranty

Restricted Rights Legend

Acknowledgments

Contents

Chapter 1: Introduction

Advanced Design System (ADS) from Agilent Technologies is a tool used by engineers for a variety of design applications, such as RFIC, System, MMIC, Hybrid or Board level design. In order to effectively use the design environment and to take advantage of its powerful simulation capabilities, designers must have a library of components that are linked to model files or simulation data.

For RFIC designers, the components and models are typically distributed by a foundry in the form of a *design kit*. A unique design kit is created for each process and each CAD tool. This kit is given to the foundry customer to use when designing their circuit.

To help RFIC designers become more successful with our electronic design automation (EDA) software, Agilent Technologies has been working with popular foundries to provide ADS components and translated model files for distribution by the foundry to the IC designer. Many customers are also creating design kits themselves.

Design kits in ADS are not only beneficial to RFIC designers. This library structure can be used for any technology or process to package and distribute a reusable set of components. With the information provided in this document, design kits for use in ADS can now be created by anyone.

The intent of this manual is to educate and assist users who are developing a design kit for the first time by providing a standard methodology. In addition to providing assistance to first time design kit developers, this manual is also intended to aid in troubleshooting problems with legacy design kits and updating these kits to the new standard.

A design kit combines functionality and features of many parts of ADS. Therefore, this document contains many cross-references to other ADS documents. Familiarity with Advanced Design System, as well as prior experience with ADS's Application Extension Language (AEL), will improve your understanding of this document and the process of creating a design kit. At a higher level, understanding this document and the concepts presented in it will guide you in the development of an integrated design flow using ADS.

The final intent of this manual is to encourage standardization of ADS design kits. This includes formalizing the structure, as well as the file formats and naming conventions, for design kits used in, but not limited to, the *ADS Front End Design Flow* and *RFIC Dynamic Link Flow*. It is absolutely imperative that your design kit follows this structure to avoid conflicts with other kits. For more information, refer to "ADS Design Flows" on page 1-2.

ADS Design Kits

An ADS Design Kit is a logical grouping of files related to a set of ADS components. The design kit structure is self-contained to provide easy transfer between different users or computer platforms. All component information needed by Advanced Design System is stored within the design kit.

At a minimum, a design kit must include a component definition file, schematic symbol files (unless built-in generic symbols are used), bitmap files for the component palette or a records file for the library browser, and information for the circuit simulator in the form of a model file, data file, or a schematic or netlisted subcircuit. Additionally, other optional files can be provided to extend the functionality of the design kit.

A design kit has a directory structure that is recognized by ADS and is similar to the directory structure for the ADS installation. All files are stored in specific directories depending on the type of file. Your ADS design kit must follow this structure, as defined in Chapter 2, Understanding the ADS Design Kit File Structure. Any deviation from this pre-defined structure can lead to serious complications with your design kit, including an inability to simulate your designs in ADS. Custom extensions within design kits can also interfere with built-in tools in ADS and cause them to fail.

The simplest way to build a design kit is to follow the tutorial steps in Chapter 3, ADS Design Kit Tutorial of this document to create a sample design kit. You can then use the information in Chapter 4, Basic Parts of an ADS Design Kit and Chapter 6, Additional Parts for ADS Design Kits to tailor your kit for your specific application. A copy of all files developed in the tutorial is provided with ADS for your convenience.

ADS Design Flows

Advanced Design System is a flexible tool that can be used on its own or in conjunction with other CAE tools in a variety of *design flows*. An engineer using the ADS *Front End Design Flow* enters a design in the ADS schematic editor and uses the ADS simulator for analysis. The design is then re-entered in a separate layout tool. To validate the integrity of the layout, the ADS Netlist Exporter is used to create a netlist for layout vs. schematic comparison. For more information on Front End Design Flow, refer to the ADS "*Netlist Exporter*" documentation.

An engineer using the *RFIC Dynamic Link Flow* enters a design in Cadence Virtuoso Schematic Capture. The design is then dynamically linked via inter-process communication (IPC) for simulation in ADS. For more information on the RFIC Dynamic Link Flow, refer to Appendix A, ADS Design Kit Development for RFIC Dynamic Link.

Design Kits versus Libraries

Before you start to build your design kit, you should make sure that a design kit is really what you need. A design kit is a complex form of a library. Advanced Design System offers other ways to create libraries of reusable parts. The simplest method is to use the **Tools > Custom Library** menu pick in the schematic window.

A *designguide* can also be viewed as a library, although the purpose is slightly different. A *designguide* is a complete study of an application topic (e.g. amplifiers, mixers, oscillators, etc.), in the form of typical simulation schematics, data displays, and detailed reference designs for study.

The following are some criteria that you can use to decide what type of library you should create.

1. Will the technology/process be used by design engineers internally or externally?
   - A design kit is better suited for external distribution.
   - For internal use, a library or *designguide* might be sufficient.
2. What are the simulation methods? (netlist, subcircuit, user-defined, data)
   - If user-defined models, data files or model cards or subcircuit models in a netlist files are being used, a design kit is recommended.
   - If a library consists strictly of schematic subcircuits, a library or *designguide* might be sufficient.
3. Will your library include custom Application Extension Language (AEL) code? AEL code is used in ADS to add parameter callbacks, layout menus and custom menus.
   - Once you decide to write custom code, it is no longer a simple library. A design kit is better suited to handle custom AEL code.
4. Will there be simulation templates?

  - Both *Design Kits* and *DesignGuides* can include simulation templates. However, the emphasis on a design kit is typically related more to the models. If providing simulation templates is your main objective, ADS *DesignGuides* are recommended as a better method.

5. What is the size of your company or the staffing model of the design department?
    - For a large company with many CAD tools and many processes to support, and a full time CAD manager supporting many designers, design kits are recommended. These can be installed in a controlled system location that is accessible by all.
    - For medium or small companies with no full time CAD manager, who just need to share subcircuits among designers, a simple library or *designguide* should be sufficient.

For more information on ADS *DesignGuides*, refer to the *“DesignGuides”* tab in your online ADS manual set.

For more information on custom libraries, refer to *“Creating Custom Libraries”* in Chapter 2 of the ADS *“Customization and Configuration”* manual.

Intended Audience

The audience intended for this manual consists of a variety of people involved in design kit creation, verification, distribution and use. This audience includes design kit developers and CAD administrators as well as advanced design kit users who need to understand what the difference is between design kits before and after ADS 2001.

What is in this Manual

This manual contains all the information needed to create a design kit for use in an *ADS Front End Design Flow*. Most of the details are applicable for all work involving design kit creation but some optional steps may be skipped for simpler kits or smaller installations. Following the standard structure will ensure that your design kit will not conflict with other design kits in use by your customers.

Included in this document are details of the directory structure, how to create the specific files needed by the system, which files are required and which are optional, and different ways to include your model information.

- Chapter 2, Understanding the ADS Design Kit File Structure, gives an overview of the directory structure of a standard ADS design kit, followed by a summary of the types of files in each directory.
- Chapter 3, ADS Design Kit Tutorial, describes in detail how to create the files in basic design kit.
- Chapter 4, Basic Parts of an ADS Design Kit, describes the fundamental parts of an ADS design kit.
- Chapter 5, Completing the Design Kit, discusses what to do after your design kit has been created.
- Chapter 6, Additional Parts for ADS Design Kits, describes the details of the parts of a design kit that were not covered in Chapter 4, Basic Parts of an ADS Design Kit.
- Chapter 7, Standardizing Existing ADS Design Kits, describes the process of standardizing design kits provided prior to ADS 2001.
- Chapter 8, Setting Up Design Kit Software and Menus, describes the details of configuration files and variables which are used to enable and disable the common design kit software.
- Appendix A, ADS Design Kit Development for RFIC Dynamic Link, includes a brief discussion of the differences between *ADS Front End Design Flow* kits and design kits using the *RFIC Dynamic Link Flow* for Cadence.
- Appendix B, ADS Design Kit Development for IFF, briefly discusses design kit development for use with intermediate file format (IFF) files.

Addressing a Needed Capability

If you are creating a design kit that needs some capability that is not included in this document or is not supported by Advanced Design System, the ADS Design Kit team at the factory would like to understand the situation. There might be a solution that has not yet been documented or we may choose to add it based on a demonstrated need and mutual, widespread benefit.

For more information, contact your Agilent Technologies sales representative with a request to submit a suggestion to the ADS Design Kit team at the factory.

Chapter 2: Understanding the ADS Design Kit File Structure

This chapter describes the details of the ADS design kit file structure. The best way to learn how to build your own design kit is to follow the tutorial steps described in Chapter 3, ADS Design Kit Tutorial. The tutorial describes how to create all the basic files and how to test the design kit as it evolves. A copy of the sample kit is provided with the ADS design kit infrastructure software. You can then use the information in Chapter 4, Basic Parts of an ADS Design Kit and Chapter 6, Additional Parts for ADS Design Kits to build on your understanding and customize the kit for your needs.

Overview of the File Structure

As mentioned in Chapter 1, Introduction, an ADS design kit is a group of files that is related to a set of ADS components, and which is self-contained for ease of transfer. The files in a design kit reside in specific subdirectories, collected under a directory that bears the name of the design kit itself. For distribution, the files are easily packaged into an archive file, in the .zip file format.

Some of the directories are required for all design kits, no matter what technology they will serve. There are also a few directories which are required depending on the technology or configuration of the design kit. Additionally, there are other directories which are completely optional and are used to provide extra functionality within the design kit.

This chapter includes an overview of all directories and the files in them, including information to help you decide which are required for your design kit. Chapter 4, Basic Parts of an ADS Design Kit gives detailed information about the required directories and files and Chapter 6, Additional Parts for ADS Design Kits gives detailed information about many of the optional directories and files. Figure 2-1 and Figure 2-2 in this chapter illustrate the structure of two simple design kits. Figure 2-3 is a comprehensive structure of all possible design kits directories defined at this time and Table 2-1 lists the types of files found in each directory.

ADS is a powerful system with capabilities for extensive customization. Design kits need to co-exist with each other and with all of the tools in the system, so they must conform to ADS standards. Agilent Technologies needs to be kept aware of the extensions that are being made for individual design kits, those that are developed by

Agilent Technologies, as well as those that are not, so that as technology advances are made, this documentation and the infrastructure software can be extended to cover new areas.

The directories and files that are required for all design kits are:

circuit/ael	Component definition
circuit/symbols	Schematic symbols
design_kit	Template ads.lib file
doc	’about.txt’ information file
examples	Schematic design file

Additionally, directories described below may be required to complete a minimal kit.

In ADS, components can be selected from a palette on the schematic page or from the library browser. The palette includes bitmaps so a component can be selected quickly. The library browser presents more information to the user and includes the ability to search for a component based on its characteristics. A component in a design kit can be presented in the palette or in the library browser or both. Additionally, a design kit may consist of some components that are only in the palette and some that are only in the library browser.

If design kit components will be available from a component palette, the following directories and files are required:

circuit/bitmaps	Bitmap files
de/ael	Boot files to enable palette

If design kit components will be available from the library browser, the following directory and files are required:

circuit/records	Library browser files

If a library browser file is included in a design kit, the .ael and .atf files in the circuit/ael directory will not need to be shipped with the design kit. However, they still need to be created so they can be compiled into the browser file. This issue will be discussed further in “Packaging for Distribution” on page 5-2.

A design kit contains information needed by the simulator to perform calculations. This information may be supplied as model cards or subcircuit models in a netlist fragment. RFIC foundry kits often include SPICE model files translated from another simulator. Completion of a design kit containing these translated model files will include a verification process that involves creating circuits in both simulators and comparing the data. The *ADS Design Kit Model Verification Toolkit* can assist you in this task. For more information on the topic of included models, refer to "Model Files" on page 4-21.

Additional topics related to inclusion of simulation data are discussed in "Adding Simulation Data to a Design Kit" on page 6-1. For example, if the model used by the design kit is a user-compiled model, you will need to supply the custom simulator executable in the *bin* directory of the design kit. Additionally, schematic subcircuits may be the form that models are delivered in. A final method of including simulation data is to include the raw simulation data in the form of an *mdif*, *s2p* or *citi* file. For more information on the *mdif*, *s2p* or *citi* file types, refer to "*Working with Data Files*" in the ADS "*Circuit Simulation*" documentation.

To supply simulation data, you will use one of the following directories:

circuit/models	netlist fragments	If simulation data is in the form of model cards or subcircuit models.
circuit/data	mdif, etc.	If simulation data is in raw data form.
bin/$ARCH	.dll (*win32*) .sl (*hpux10*/*aix4*) .so (*sun57*)	If a user-compiled model is included as a dynamically linked library (.dll) or shared library (.sl or .so).
circuit/designs	.dsn files	If a subcircuit model is in schematic form.

Note $ARCH can be determined by running $HPEESOF_DIR/bin/hpeesofarch. For ADS2002, the ARCH values *win32*, *hpux10*, *aix4*, and *sun57* are returned from this program. They may change in the future.

Figure 2-1 shows the directory structure of a design kit that provides a component palette and model data in the form of model cards or subcircuit models in a netlist file. This is a typical simple RFIC design kit. Additionally, it could include a circuit/records directory to enable the library browser.

The name *design_kit_name* as the top level directory name is a generic name supplied for this illustration. This name should be replaced by the actual name of your design kit.

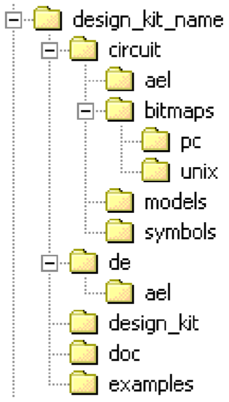


Figure 2-1. ADS Design Kit File Structure for Component Palette and Simulation Data in Model Files

Figure 2-2 shows the directory structure of a design kit that provides access to components via the library browser and includes simulation data in the form of a data file.

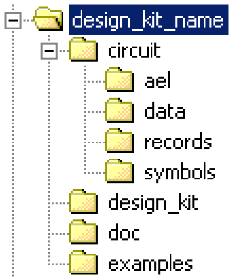
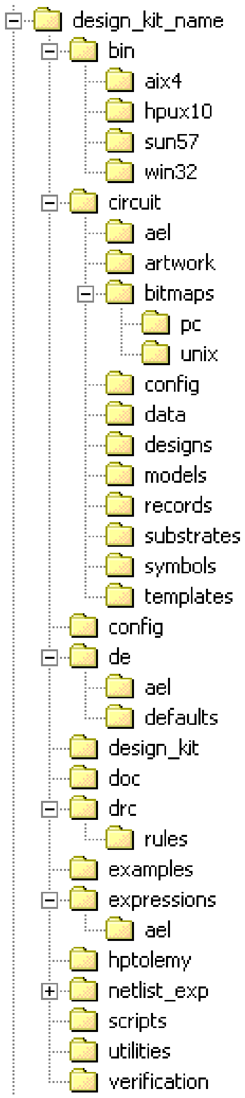


Figure 2-2. ADS Design Kit File Structure for Library Browser Access and Simulation Data in Data Files

Figure 2-3 shows the comprehensive structure of all possible design kit directories defined at the time this document was written. If your design kit needs directories that are not listed in the complete structure shown here, it is recommended that you work with the ADS Design Kit Infrastructure team at Agilent EEsof-EDA.



```
design_kit_name
    bin
        aix4
        hpux10
        sun57
        win32
    circuit
        ael
        artwork
        bitmaps
            pc
            unix
        config
        data
        designs
        models
        records
        substrates
        symbols
        templates
    config
    de
        ael
        defaults
    design_kit
    doc
    drc
        rules
    examples
    expressions
        ael
    hptolemy
    netlist_exp
    scripts
    utilities
    verification
```

Figure 2-3. Comprehensive Listing of Potential Directories in an ADS Design Kit

Table 2-1 lists all of the possible subdirectories, as shown in Figure 2-3, as well as the types of files that will reside in those directories. Note that the last column in Table 2-1 denotes whether the files are autoloaded or not.

Table 2-1. Design Kit File Structure

Directories	Subdirectories	Files	Description	AL
bin/	$ARCH/	*.dll (win32), *.sl (hpux10 or aix4), *.so (sun57) †	User-compiled model in dynamically linked library or shared library.	Y
circuit/	ael/	<prefix>_<item>.ael	create_item()	Y*
	artwork/	*.ael	macro files for ADS layout	N
	bitmaps/pc/	<prefix>_<item>.bmp	PC bitmap	N
	bitmaps/unix/	<prefix>_<item>.bmp	UNIX bitmap (bmptoxpm)	N
	config/ †	ADSlibconfig	#uselib lookup table	N
	data/	*.ds	simulations/measurements	Y
		*.mdf	MDIF files generated by ICCAP	Y
		*.s2p	Touchstone S-parameter files	Y
		*.cti	CITIFILE files	Y
	designs/	*.dsn	subcircuits	Y
	models/	*.net	spice/spectre translator netlists	N
	records/	<prefix>_<lib>.ctl	control file	Y
		<prefix>_<lib>.rec	record file	Y
		<prefix>_<lib>.idf	item definition file (hpedlibgen)	Y
	substrates/ †	*.slm	Momentum substrate file	N
	symbols/	<prefix>_<item>.dsn	symbols	Y
	templates/ †	*.ddt, *.rec, *.ctl	dds templates	Y
		*.tpl	schematic templates	Y
config/		de_sim.cfg †	template config files	N
de/	ael/	boot.ael	generic	Y*
		palette.ael	generic	N
	defaults/ †	*.lay	layer files	N
		*.prf	preference files	N
design_kit/		ads.lib	template	Y

Table 2-1. Design Kit File Structure

doc/		readme.txt	install info	N
		about.txt	menu info	N
		index †	lookup file	N
		<prefix>_<lib>.html †	component info	N
drc/	rules/ †		DRC rules	N
examples/		*.zap	archived ADS project	N
expressions/	ael/ †	expressions_init.ael	expression load file	N
		*.ael	expressions	N
hptolemy/ †			hptolemy dir	N
lvs/ †	ael/		layout vs. schematic	N
	components/			N
	config/			N
scripts/ †		*.pl	scripts	N
		*.ksh, *.bat		N
utilities/ †		*.ael	any auxiliary code	N
verification/ †			verification info	N

AL = Files Autoloaded (Y/N)? Note that <prefix>_<item>.ael is loaded only if specified in boot.ael or if compiled into <prefix>_<lib>.idf. Also, boot.ael is loaded only when specified in ads.lib.

† Optional part. For more information, refer to Chapter 6, Additional Parts for ADS Design Kits.

Understanding the Directory Contents

This section briefly describes the files in each subdirectory of the complete ADS design kit structure. For a full description of each section, refer to the expanded discussion in Chapter 4, Basic Parts of an ADS Design Kit or Chapter 6, Additional Parts for ADS Design Kits.

The naming convention for some of the items in these tables is <*prefix*>_<*item*>.ael or <*prefix*>_<*lib*>.ael. The <*prefix*> name is a unique identifier that should include the foundry name and the process name. The <*item*> name refers to a component name and the <*lib*> name refers to a library name. For more information on naming conventions, refer to "Component Name" on page 4-2. Note that all names within angled brackets (< >) are place holders and will be replaced by a real name in your design kit.

The '<design_kit_name>' Directory

The <*design_kit_name*> directory is the ADS Design Kit directory name. For more information on naming directories, refer to "Design Kit Name" on page 4-1.

The 'bin/$ARCH' Directory

The bin/$ARCH directory contains platform dependent subdirectories, where $ARCH = *hpux10*, *aix4*, *sun57*, or *win32* for ADS2002. The directories contain dynamically linked libraries or shared libraries (*.dll/*.sl/*.so). The value of $ARCH can be determined by running the command:

```
$HPEESOF_DIR/bin/hpeesofarch
```

For more information, refer to "User Compiled Models" on page 6-2.

The 'circuit' Directory

For each design kit, there are quite a few possible *circuit* subdirectories that contain the files specific to a particular process. Details on the files in these subdirectories, are provided in Table 2-2.

Table 2-2. The *circuit* Subdirectories

Subdirectory	Description
ael	Component definition files The files in these directories are AEL files containing 'create_item()' statements, which are the item definitions for each component. Each component may be in a separate file or they may be combined into one file. The file may also contain global variable declarations, form definitions, or callback functions needed for each component. These files are loaded automatically only if there is no boot file specified in ads.lib and no .idf file in the circuit/records directory. Files: <*prefix*>_<*item*>.ael
artwork	Macro files for Component Artwork This directory is provided for AEL artwork macro files for ADS layout. Files: *.ael

Table 2-2. The *circuit* Subdirectories

Subdirectory	Description
bitmaps	Component bitmap files Each component has two versions of the same bitmap stored in two separate subdirectories. The file for the PC platform is in BMP format, and the file for the unix platform is in the XPM format. The file names should be the same between the two directories and should be stored in subdirectories named pc and unix. All files can have the .bmp extension, which cannot be given when the file is referenced in ael calls to de_define_palette_group(). Files: *<prefix>_<item>*.bmp
config †	Component configuration files The ADSlibconfig file uses the #uselib lookup table.
data	Component data files These files are an optional way to define simulation data. The typical way is to provide model files. This directory is provided to store ADS simulator dataset files (*.ds), MDIF files (*.mdf), CITIFILES (*.cti) and Touchstone files (*.s2p). Files: *.ds, *.mdf, *.cti, or *.s2p
designs	Component design files The *designs* directory contains the design files (*.dsn) for hierarchical subcircuits that can be pushed into from the schematic view. Files: *.dsn
models	Component model files This is the typical way to provide model data to the simulator. It is a file or set of files containing process variables and model cards or subcircuit model descriptions in the ADS netlist format. Typically these files are translated from Spectre or HSpice files with the ADS Netlist Translator. For more information on netlist translation, refer to the ADS Netlist Translator documentation. Files: *.net

Table 2-2. The *circuit* Subdirectories

Subdirectory	**Description**
records	Component records files There are three types of files in the records directory. The .ctl file lists the names of the libraries in an XML file format. It also includes the name of a .rec file to read for each library. The .rec file is also in XML format and it lists specific information about each component in the library. These two files are used to build the library and component lists in the Library Browser. Only one .ctl file should be included and it can refer to multiple .rec files. The third type of file in the directory is used for demand loading of components. It is a platform-independent hash file created from all the .ael item definition files for each component. Only one .idf file should be included in this directory. The records directory is added to the DESIGN_KIT_BROWSER_PATH search path when a design kit is loaded. This variable is referenced by HPANALOGRF_BROWSER_PATH in hpeesofbrowser.cfg. Files: *<prefix>_<lib>*.ctl; *<prefix>_<lib>*.rec; *<prefix>_<lib>*.idf
substrates †	Component substrate files Each component has a location for Momentum substrate files. For more information on Momentum substrate files, refer to the “Momentum” documentation. The design software does not currently contain any functionality tied to these directories, however, specific design kits may include custom software that uses them. Files: *.slm
symbols	Component symbol files The symbol files are design files containing the information needed by the system to draw the schematic symbol on the schematic page. The configuration variable SYSTEM_CUSTOM_CIRCUIT_SYMBOLS is modified on the fly when a design kit is loaded. The path is extended to include the path to these symbol files. Files: *<prefix>_<item>*.dsn

Table 2-2. The *circuit* Subdirectories

Subdirectory	Description
templates †	Component template files This directory can include templates for simulation or for data display. The path variable DESIGN_KIT_TEMPLATE_BROWSER_PATH is extended to include this directory when a design kit is loaded. This path variable is referenced by HP_TEMPLATE_BROWSER_PATH in hpeesofbrowser.cfg. Files: <*prefix*>_<*item*>.ds or <*prefix*>_<*item*>.mdf
† Optional part. For more information, refer to Chapter 6, Additional Parts for ADS Design Kits.	

The 'config' Directory

The <*design_kit_name*>/config is no longer required. It was used in the past to store a template version of the *de_sim.cfg* file that needed to be placed in the $HOME/hpeesof/config directory at the local level or $HPEESOF_DIR/custom/config directory at the system level. The variables defined in this file were necessary to load the old design kit software. This is no longer necessary as of ADS 2001. For more information on the *de_sim.cfg* file, refer to "Configuration Files" on page 8-1.

This directory may be used if a kit requires custom configuration variables but extensive use of these variables is discouraged. A script would have to be provided to extend configuration variables dynamically or move the file to a location where it will be recognized by ADS, and the script has to be smart enough to merge the file with existing files and specific variables with existing variables of the same name to avoid disabling other software. Details of this level of customization are not currently provided in this documentation. For more information on custom configurations, refer to the ADS "*Customization and Configuration*" documentation.

The 'de' Directory

For each design kit, there are two possible *de* subdirectories that contain the files specific to a particular process. Details on the files in these subdirectories, are provided in Table 2-3.

Table 2-3. The *de* Subdirectories

Subdirectory	Description
ael	AEL files The <*design_kit_name*>/de/ael directory is to be used for AEL files that apply to the design kit in general, as opposed to the component specific files in circuit/ael. Some examples of files in this directory are *boot.ael* and *palette.ael*. These files contain commands used by the system to load the design kit and configure the component palette. More information about these files is included in Chapter 4, Basic Parts of an ADS Design Kit. Sample versions of these files are included in the Chapter 3, ADS Design Kit Tutorial. Other AEL files may be put in this directory if the design kit requires custom AEL code. The utilities directory is also available for auxiliary AEL functions. ADS must be instructed to load these AEL files from *boot.ael*, as shown in "Creating the boot.ael File" on page 3-5, and *boot.ael* must be specified in the template *ads.lib* file. Files: <*prefix*>_<*item*>.ael
defaults	Layer and Preferences files The <*design_kit_name*>/de/defaults directory can contain layer and preference files (.lay and .prf) which need to be used with the design kit. If the de/defaults subdirectory is included in a design kit, the design kit needs to include custom AEL for handling these files. Details on the possible methods for handling this are included in "Layers and Preferences Files" on page 6-17. For more information on layer (.lay) and preference (.prf) files, refer to "*Preference Functions*" in the "*AEL*" documentation. Files: *.lay, *.prf

The 'design_kit' Directory

The <*design_kit_name*>/design_kit directory contains a template version of the *ads.lib* file that needs to be installed to enable this design kit. For specific details on the ads.lib file, refer to "The ads.lib Template" on page 4-33.

The 'doc' Directory

The <*design_kit_name*>/doc directory contains all of the user documentation associated with the design kit. The only required file in the doc directory at this time is the *about.txt* file. HTML files may also be included and merged into the ADS documentation set. For more information on the *about.txt* file, refer to "The about.txt File" on page 4-36. For more information on adding HTML documentation to the system, refer to "Creating Design Kit Documentation" on page 6-16.

The 'drc' Directory

The <*design_kit_name*>/drc/rules directory contains ael files which define drc rules for the Design Rule Checker (DRC) tool in ADS. There is no automatic process at this time for the ADS system to recognize the drc files in the design kit directory so custom AEL must be provided to copy the file to an ADS project directory. A custom menu pick can be provided to facilitate this. For more information on DRC in ADS, refer to the "*Design Rule Checker*" documentation.

The 'examples' Directory

Each design kit must include at least one sample design. This can be used by your customer or by Agilent Technologies support engineers to ensure that the design kit is installed correctly before they start a new design. You can use this as a method to demonstrate special features of the design kit such as a mandatory process component that must be placed in any design using the design kit. It is especially helpful if a design kit includes a custom version of the simulator to verify that the correct simulator is being used. For more information, refer to "Adding Simulation Data to a Design Kit" on page 6-1. The project or projects containing the sample design(s) are archived and then stored in the <*design_kit_name*>/examples directory.

The 'expressions' Directory

The <*design_kit_name*>/expressions/ael directory contains files which contain expressions that can be copied into a schematic or data display for processing before or after simulation. For more information, refer to "Expressions" on page 6-18.

The 'hptolemy' Directory

The <*design_kit_name*>/hptolemy directory is not currently used by standard design kit software, however, it may be used by specific or custom design kit software.

The 'netlist_exp' Directory

The <*design_kit_name*>/netlist_exp directory contains all files needed by the ADS *Netlist Exporter*. By providing rules files for each component in a design kit, the Netlist Exporter can output netlists in the proper form for many LVS tools. For more information on incorporating the appropriate LVS information into a design kit, refer to "Layout vs. Schematic Comparison" on page 6-16.

The 'scripts' Directory

The <*design_kit_name*>/scripts directory contains any custom shell or perl scripts that you have developed for use with your design kit. For more information on incorporating these scripts into your design kit, refer to Chapter 6, Additional Parts for ADS Design Kits.

The 'utilities' Directory

The <*design_kit_name*>/utilities directory contains any custom AEL scripts that are auxiliary in nature. Other AEL scripts that are used directly with the design kit are stored in de/ael or circuits/ael as described in "The 'de' Directory" on page 2-12 and "The 'circuit' Directory" on page 2-9. For more information on incorporating these scripts into your design kit, refer to Chapter 6, Additional Parts for ADS Design Kits.

The 'verification' Directory

The <*design_kit_name*>/verification directory contains files that are used by the *ADS Design Kit Model Verification Toolkit* to create a verification suite. These files are appropriate to ship along with the design kit. For more information on the verification tool, refer to "Verifying a Design Kit" on page 5-1.

Chapter 3: ADS Design Kit Tutorial

This chapter provides step-by-step instructions for creating the basic parts of a sample design kit consisting of several typical RFIC foundry kit components:

- A device with model reference
- An include component
- A device with subcircuit model
- A device with Symbolically Defined Device (SDD) reference

Even though these components may not be typical for all design kits, the steps are applicable to a design kit in any technology.

To complete this tutorial, it is assumed that you have a basic working knowledge of Advanced Design System, including the location of the ADS installation directory for your computer or site. This tutorial will refer to your ADS installation directory, which is defined as $HPEESOF_DIR. For example, on a PC, an ADS 2002 installation is typically installed in C:/ADS2002. If you cannot find the installation directory on your computer or the site-wide installation on a networked system, contact your system administrator or CAD manager. For simplicity, the tutorial steps will assume a typical PC installation.

Note The directory delimiter slash is shown as a forward slash (/). This should be used in most cases in a design kit, especially in AEL files, where the backslash character (\) is interpreted as a string formatting character. For example, “\n” is interpreted as a new line character. The AEL **system()** function is one of the rare cases where the back slash character may be required, and each backslash must be preceded by an extra backslash to tell the system to interpret it literally. The list of known characters that cause this problem are:

\n=new line
\r=return
\f=form feed
\b=back space
\t=tab

Tutorial Overview

The list below shows the basic steps that will be performed in this tutorial and includes a short description of each section so you can quickly get a sense of what is required to build a simple design kit.

Note The completed sample design kit is available with the ADS design kit software. For more information, refer to "Accessing the Supplied Sample Kit" on page 3-46.

- "Creating the ads.lib File" on page 3-4 describes how to create the file (ads.lib) that contains the design kits that will be loaded.
- "Creating the boot.ael File" on page 3-5 describes how to create a boot file (boot.ael) to load the design kit.
- "Creating Component Symbols" on page 3-7 explains how to create a schematic symbol (*.dsn) for each component in your design kit.
- "Creating Component Definitions" on page 3-9 discusses how to build a component definition AEL file (mykit_item.ael) which defines how the component is netlisted along with other properties.
- "Testing Your Component" on page 3-12 lists a few steps you can use to verify that your components are working properly.
- "Providing Basic Documentation" on page 3-13 gives an example for creating a simple text file (about.txt) used to document the design kit.
- "Making Components Accessible" on page 3-14 simply describes the two methods available for setting up easy access to your design kit components.
- "Creating a Component Palette and Bitmaps" on page 3-14 describes how to create bitmaps (*.bmp) for your design kit components. The AEL file (palette.ael), which loads the bitmaps onto a palette and makes the palette available in an ADS schematic window, is also described in this section.
- "Adding a Netlist Include Component" on page 3-18 describes how to create a netlist include component which can be used to include model files (*.net) in your netlist.
- "Creating an Example Design Using your Design Kit" on page 3-23 shows an example schematic that uses the design kit created in the tutorial.

- “Adding Components to the Library Browser” on page 3-26 describes two methods for making your design kit components visible in the ADS library browser.
- “Adding Demand Loaded Components” on page 3-29 describes how to create the item definition file (*.idf) used for dynamically loading design kit components.
- “Using a Subcircuit Model” on page 3-31 provides an outline of how to include a subcircuit model in your design kit using the information learned in the tutorial.
- “Adding a Resistor with SDD Subcircuit Model” on page 3-36 describes another component that uses a subcircuit model in a netlist file. It is included in the tutorial to show a simple example of using a Symbolically Defined Device (SDD) to define arbitrary current/voltage relationships.

Building the Basic Design Kit Parts

The first step in creating a design kit is to create the directory in which to store the basic design kit parts after they have been built. This will be the directory that you will eventually archive and distribute as the design kit. Create that directory now. You can create this directory anywhere but to simplify the tutorial, it is assumed that it will be created in the home directory ($HOME). Name this directory *my_design_kit*.

Create the following subdirectories in *my_design_kit*.

```
  design_kit
  doc
  circuit/symbols
  circuit/ael
  circuit/models
* circuit/bitmaps/pc
* circuit/bitmaps/unix
* circuit/records
  de/ael
  examples
```

Note Directories marked with an asterisk (*) above are optional directories that are used in this tutorial.

Creating the ads.lib File

As you will learn in "The ads.lib Template" on page 4-33, the *ads.lib* file contains the information that tells ADS which design kits to load and where to get the specific instructions for loading each kit.

To create the my_design_kit/design_kit/ads.lib file that will be used in this tutorial:

1. Open a text editor.
2. Enter the following line, *exactly as it is written here*, in a text file.

   ```
   MY_DESIGN_KIT | path_to_design_kit_directory | de/ael/boot | mykit_v1
   ```

3. Save the file as:

   $HOME/my_design_kit/design_kit/ads.lib

   Note that this assumes that you have created the *my_design_kit/design_kit* subdirectories under your $HOME directory as described in "Building the Basic Design Kit Parts" on page 3-3.

This is now the template ads.lib file described in Chapter 2, Understanding the ADS Design Kit File Structure and "The ads.lib Template" on page 4-33. The path cannot be entered at this time since it requires knowledge of where the design kit will be installed on the end user's system.

The design kit software will set the path after the design kit has been installed and when the ads.lib file is being copied to a predefined directory on the user's system. You will do this manually in the next step for your own testing.

For testing the design kit, this information will need to be available in the user customization directory for design kits, $HOME/hpeesof/design_kit.

1. Determine if there is already a file named ads.lib in $HOME/hpeesof/design_kit.
2. If there is not, simply copy the ads.lib file created in the last step to that location. If there is already a file by that name, add the contents of the new file to the existing file.

3. Further edit this ads.lib file to set the actual design kit directory. For this tutorial, that directory is c:/my_design_kit on a PC. This will be different on unix.
4. Save the ads.lib file and close the text editor.

Creating the boot.ael File

An ael file called *boot.ael* must be provided so that ADS knows how to load the design kit. From the boot.ael file, you can load other AEL files, such as those that will load the item definition and palettes as described in later sections.

To create the de/ael/boot.ael file:

1. Open a text editor.
2. Copy the text in Table 3-1 into a new file. The sample boot.ael provided in Table 3-1 includes more information than is required. This is provided to teach you about printing debug information from an AEL file.

Table 3-1. The Boot File (*boot.ael*)

```
// boot.ael - This file resides in the de/ael directory of the design kit.
// It is loaded by the design kit infrastructure software if it is listed
// in the file ads.lib in one of 4 predefined locations, one of which is
// $HOME/hpeesof/design_kit. This file is used to load other AEL files
// such as palette.ael. It is also used to set up some global variables for
// use in other files.
// Some global variables that are available by default are:
// designKitRecord - this is a list which contains the 4 records from
//                   ads.lib (kit name, path, boot file, version).
// As soon as the design kit load process has finished, this variable is
// unset, so save the values as a variable with a different name if you
// want access to them later.
//
// MY_DESIGN_KIT - this is the first record from ads.lib. Its value is the
//                 second record from ads.lib, the path to the design kit.
// The following debug print statements can be used To view the values of
// these variables:
//
// To print a field in the list:
fputs(stderr, designKitRecord[0]);

// To view the special kit name/path variable. This name will change
// depending on the name of the kit as registered in the first field of the
// ads.lib entry

fputs(stderr, MY_DESIGN_KIT);

// Comment out all debug print statements before shipping your design kit.

// These path names will be used later to load other files.
decl MYKIT_BITMAP_DIR = sprintf("%s/circuit/bitmaps/%s/", MY_DESIGN_KIT,
                        on_PC?"pc":"unix");
decl MYKIT_CIRCUIT_AEL_DIR = sprintf("%s/circuit/ael/", MY_DESIGN_KIT);
decl MYKIT_CIRCUIT_MODEL_DIR = sprintf("%s/circuit/models", MY_DESIGN_KIT);
decl MYKIT_DE_AEL_DIR = sprintf("%s/de/ael/", MY_DESIGN_KIT);
// To print a variable:
fputs(stderr, MYKIT_BITMAP_DIR);
```

3. Save the file as $HOME/my_design_kit/de/ael/boot.ael

Table 3-1 shows the contents of your new boot.ael file. This code will be added to later in this tutorial. By accessing certain global variables, you can get the values from the ads.lib file, as described in“Creating the ads.lib File” on page 3-4. These values are stored in a list called *designKitRecord*. Debug tips are included directly in the boot.ael code for your convenience. Remember to remove all debug print statements before you ship your design kit.

Important Variables such as those in Table 3-1 called MY_DESIGN_KIT, etc., are global variables. The scope of an AEL variable is not restricted to the file in which it is defined. In other words, this variable must be unique in the system to avoid being overwritten by another declaration of the same variable. Therefore, in the future when you build design kits, make sure the name “MY_DESIGN_KIT” in *boot.ael* is replaced by the actual name of your kit, which is the first field in the *ads.lib* file.

Viewing Debug Output

On unix, debug output is visible in the window from which ADS was invoked. On PC, edit the *shortcut* property *Target* and add **-d daemon.log** to the end of the command line (Example: C:/ADS2002/bin/hpads.exe -d daemon.log). This will enable you to view output to stderr in a log window. Look for text in red. All other text is a communication log. The information is also saved to a file called *daemon.log* in the ADS startup directory.

Creating Component Symbols

The next part of your design kit to be created is a schematic symbol for each of your components. The tutorial will guide you through one method of symbol creation, copying an existing symbol. Chapter 4, Basic Parts of an ADS Design Kit describes two other methods in “Schematic Symbol” on page 4-12.

Before creating your symbols, start ADS and create a new project called *my_kit_prj* in which to build the component symbols. You can create this project anywhere, but to simplify the tutorial, it is assumed that the project will be created in your home directory.

To copy an existing symbol:

1. Open a schematic page in your new project.
2. In the schematic window, choose **View > Create/Edit Schematic Symbol**. The *Symbol Generator* dialog box appears.
3. In the Symbol Generator dialog box, select the *Copy/Modify* tab.

4. Set the Symbol Category to *Devices-BJT*.
5. Click the *BJT_NPN* symbol so the SYM_BJT_NPN appears in the Symbol Name field, then click **OK**.

Note There is a warning about symbol pin/component port mismatch - this can be ignored.

---

6. Once you have copied the built-in symbol, you can edit the symbol if desired. However it will not be modified for this example.
7. Choose **File > Save Design As** and save the symbol file as a new symbol called *SYM_mykit_npn.dsn*.

You should now have a symbol for use in your design kit. The file containing this symbol was saved in the networks directory of the project as a .dsn file. You may also see an .ael file with the same name, which will not be used.

At this time, you should delete any .ael files that may be present and copy the symbol file into the proper subdirectory of the design kit directory. That directory is $HOME/my_design_kit/circuit/symbols.

Creating Component Definitions

A component definition is added to the design environment by a **create_item()** command provided in an AEL file. The **create_item()** command is very complex and is documented in Chapter 15 of the *"AEL"* documentation.

Chapter 4, Basic Parts of an ADS Design Kit also includes more information about the **create_item()** function. For this tutorial, the most you need to understand is how to define a symbol, a list of parameters, and how the component is netlisted. These are all parts of the item definition.

The **create_item()** functions for your sample components should be defined as follows in a file called *mykit_item.ael*. For a small design kit, all the item definitions could be combined into one file like this. Another acceptable method is to create a separate file for each item. This is especially recommended if there are parameter callbacks associated with each component. In this case, a file called *mykit_npn.ael* would be appropriate. It is, however, acceptable to leave all components in one *mykit_item.ael* file, which is what the tutorial will instruct you to do.

Since the **create_item()** command is somewhat complex, you might find it helpful to let the system start to create your component definition for you. You can follow the steps below, which will guide you through creating and modifying the file, or you can skip steps 1-10 and just copy and paste the code shown in Table 3-2. Save the file as *mykit_item.ael* in the circuit/ael directory of your design kit and then perform step 11, which instructs the design kit to load the item definition.

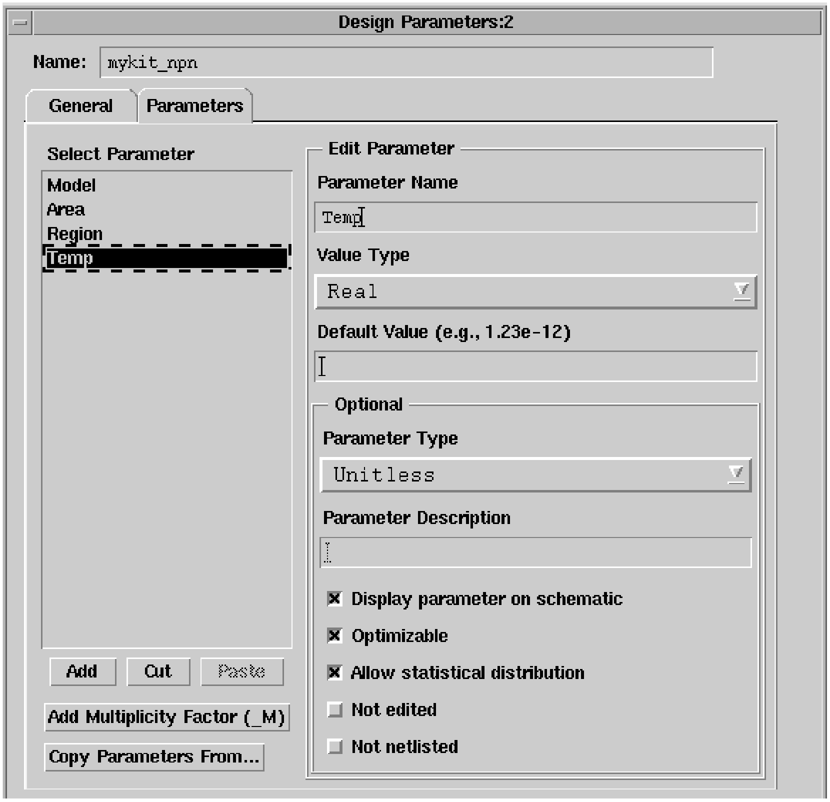
To create a component definition for your BJT component:

1. From the ADS Main window, choose **File > New Design** to open a new design in ADS. The New Design dialog box appears.
2. In the New Design dialog box, name the design *mykit_npn*. Click **OK** to close the New Design dialog. A new schematic window is generated with the title *mykit_npn*.
3. From the new schematic window, choose **File > Design Parameters**. The Design Parameters dialog box appears.
4. In the **General** tab, enter the following information:
    - Description: MYKIT Nonlinear Bipolar Transistor, NPN
    - Component Instance Name: BJT
    - Symbol Name: SYM_mykit_npn

- Library Name: MY_DESIGN_KIT
- Model: Built-in Component
- Artwork Type: None

Accept the defaults for all other fields.

5. In the **Parameters** tab, enter the following Parameter Names and then click the **Add** button. Repeat for all parameters listed below.
   - Model
   - Area
   - Region
   - Temp


Design Parameters:2

Name: mykit_npn

General | Parameters

Select Parameter
- Model
- Area
- Region
- Temp

Edit Parameter
Parameter Name: Temp
Value Type: Real
Default Value (e.g., 1.23e-12):

Optional
Parameter Type: Unitless
Parameter Description:

- [✓] Display parameter on schematic
- [✓] Optimizable
- [✓] Allow statistical distribution
- [ ] Not edited
- [ ] Not netlisted

Note The Design Parameter dialog box contains a parameter option check box labeled *Not Edited*. This is used if a parameter value never needs to be modified by the user. However, it will also make the parameter un-editable by callbacks as well, so do not set that flag if the parameter needs to be modified by a parameter callback.

6. Click **OK** in the Design Parameters dialog to save and exit from the dialog.
7. Save the design, without entering anything in the schematic window.
8. Find the AEL file which was saved with the item definition. It will be in the networks directory of the project and will be called *mykit_npn.ael*.
9. Copy *mykit_npn.ael* to your design kit circuit/ael directory and rename it *mykit_item.ael*.
10. Edit the file so that it looks the same as the file in Table 3-2. Also, delete the last 3 lines of the file.
11. In order for ADS to read the new *mykit_item.ael* file just created, the boot file must be modified to load the *mykit_item.ael* file. Add the following line to the end of the boot.ael and save the boot.ael file.

```
load(strcat(MYKIT_CIRCUIT_AEL_DIR,"mykit_item"), "CmdOp");
```

Note When writing AEL code, you must be very careful that each quote, semi-colon, parenthesis and bracket is exactly as shown. The AEL interpreter will fail if there are errors in the file, and it may not tell you where the errors are. If you get error messages when starting ADS, check your file against the file in Table 3-2 or compare it against the tutorial file shipped with the software.

Table 3-2. The Item Definition File (*mykit_item.ael*)

```
set_simulator_type(1);
create_item("mykit_npn",
          "MYKIT Nonlinear Bipolar Transistor,NPN",
          "BJT",NULL, NULL, NULL,
          standard_dialog, "",
          CmpModelNetlistFmt, "",
          ComponentAnnotFmt,
          "SYM_mykit_npn",
          no_artwork, NULL,
          ITEM_PRIMITIVE_EX,
      create_parm("Model", "Model instance name", 0,
          "StdFileFormSet",UNITLESS_UNIT,prm("StdForm","BJTM1")),
      create_parm("Area","Scaling Factor, (default: 1.0)",
          PARM_OPTIMIZABLE | PARM_STATISTICAL,
          "StdFileFormSet",UNITLESS_UNIT,prm("StdForm","")),
      create_parm("Region",
          "DC operating region, 0=off, 1=on, 2=rev, 3=sat, (default: on)",
          0, "StdFileFormSet",UNITLESS_UNIT,prm("StdForm","")),
      create_parm("Temp", "Device operating temperature, (default: 25)",
          PARM_OPTIMIZABLE | PARM_STATISTICAL,
          "StdFileFormSet",TEMPERATURE_UNIT,prm("StdForm","")));
```

Testing Your Component

You will now be able to test your first component if you have set up all the files defined so far in this chapter. The files shown in Table 3-3 should now be saved in their respective directories under the design kit top level directory ($HOME/my_design_kit).

Table 3-3. Tutorial Directory and File Locations

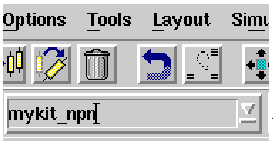
Directory	File(s)
design_kit	ads.lib (plus a copy of it in $HOME/hpeesof/design_kit/ads.lib)
de/ael	boot.ael
circuit/symbols	SYM_mykit_npn.dsn
circuit/ael	mykit_item.ael

To test your new component:

1. Restart ADS to automatically pick up the new files.

2. From the ADS *Main* window, choose **DesignKit > List Design Kits**. The List ADS Design Kits dialog box appears. Your design kit should be listed in the *Name* column of the dialog and the *Status* should be *enabled*.

3. Open a schematic window and type the name of your component in the Component History dialog.



Component History Drop-down List

4. After entering the component, locate your cursor onto the schematic page and place your component.

If there are no apparent problems after performing the steps listed above, your new design kit component should be ready to use.

Providing Basic Documentation

Using a text editor, you will now create the *about.txt* file to document your new design kit. This file should include the information listed in the template in Table 4-6. Note that the template is only a suggested format for the file. You may modify the format of the file or include information not listed in the template.

To create the my_design_kit/doc/about.txt file:

1. Open a text editor.

2. Add the following information to a file:

```
Name: MY_DESIGN_KIT

Version: 1.0

Date: 11/15/2001

Description: This design kit contains the components mykit_npn and
mykit_include.

Revision History: Rev. 1.
```

3. Save the new file as $HOME/my_design_kit/doc/about.txt

Making Components Accessible

There are two methods for accessing your design kit components:

- A component palette on the schematic window
- The ADS Library Browser

The component palette is on the left border of the schematic window. The library browser can be opened by choosing **Insert > Component > Component Library**. The default tool bars also include a *Display Component Library List* button to open the library browser.

Library control and records files are used to load the necessary information into the library browser.

While this tutorial will show you how to setup both the component palette and the library browser, you may decide to only provide one or the other. For more information on choosing one of the two, refer to "Component Palette vs. Library Browser" on page 4-15.

Creating a Component Palette and Bitmaps

Defining a component palette consists of two steps:

- Creating two bitmap files (one for pc and one for unix) for each component.
- Writing an AEL function to connect the bitmap to the component and load the definition into the system.

Chapter 4, Basic Parts of an ADS Design Kit, discusses the details of bitmap creation for design kits. A new tool offered within ADS is the *DesignGuide Developer Studio*. The Developer Studio can be installed from your ADS installation CD, if it is not already installed, and it does not require a license to run. A design guide is different than a design kit, but the bitmap tool can be used for both. For more information on the differences between DesignGuides and design kits, refer to "Design Kits versus Libraries" on page 1-3. This tool is mentioned here because it provides a bitmap editor which is specially designed to create bitmaps for use in ADS. The **Save As UNIX** and **Save As PC** menu picks facilitate saving the bitmaps in both formats. Sample bitmaps for all ADS components are available with the tool to copy as starting material. For an explanation of how to use the various features of the tool, refer to *Chapter 5* of the *"DesignGuide Developer Studio"* documentation.

For a design kit with a large number of components, more than one palette group may be defined. For the sample kit in this tutorial, all components will go into one palette.

To create your UNIX and PC bitmaps:

1. From the ADS *Main* window, choose **DesignGuide > DesignGuide Developer Studio > Start DesignGuide Studio.** The *DesignGuide Developer Studio* dialog box appears. A warning dialog may appear; however, this does not apply to using just the bitmap editor so close the warning dialog box and proceed with step 2.
2. From the Developer Studio dialog, choose **Tools > Bitmap Editor**. The *Bitmap Editor* dialog box appears.
3. From the Bitmap Editor, choose **File > Open**. The *Open Bitmap* dialog box appears.
4. Sample bitmaps are provided in:

   $HPEESOF_DIR/designguides/projects/dgstudio/ui/bitmaps/adsbmps

   Open the bitmap for *BJTNPN.BMP* and edit the bitmap as needed to customize it for your design kit (see Figure 3-1).

Note On unix, change the Filter in the Open Bitmap dialog box to capital *.BMP to see the BJTNPN.BMP file.

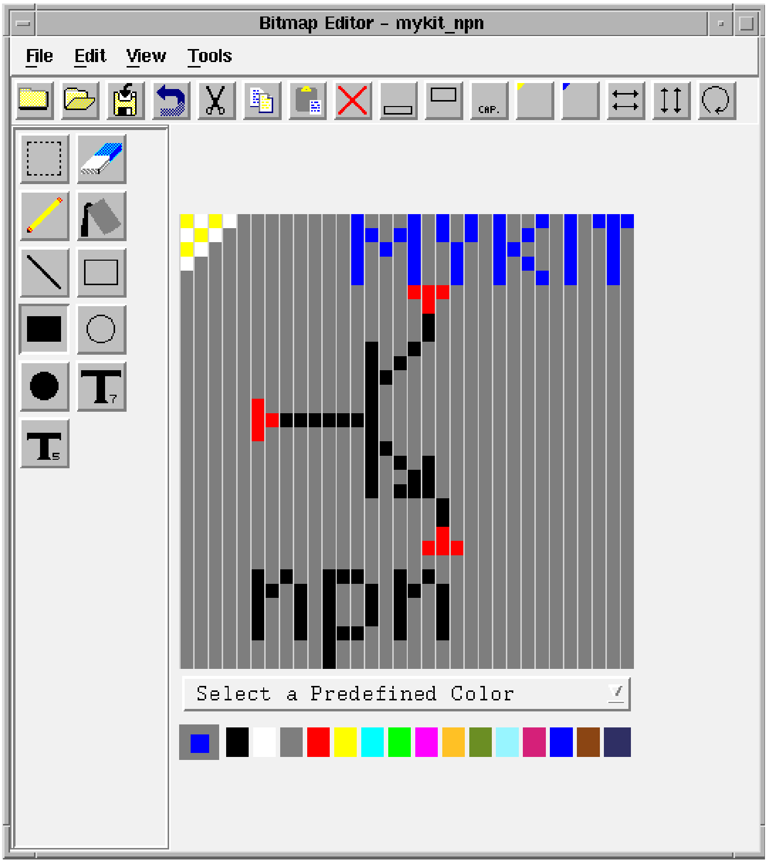


Figure 3-1. The *mykit_npn.bmp* File in the Bitmap Editor

5. After editing your bitmap file, save a UNIX and a PC version of the bitmap in the appropriate design kit directories. Remember that these are:

   - my_design_kit/circuit/bitmaps/pc
   - my_design_kit/circuit/bitmaps/unix

   Name the BJT bitmap *mykit_npn.bmp*

The final step in making the palette available is to create an AEL file called *palette.ael*, which contains a function call to the ADS function, **de_define_palette_group().** To create the de/ael/palette.ael file:

1. Open a text editor.
2. Copy the text in Table 3-4 into a file.

Table 3-4. The *palette.ael* File

```
de_define_palette_group(SCHEM_WIN, "analogRF_net",
            "MYKIT Components","MYKIT Components", 0,
            "mykit_npn", "MYKIT NPN Bipolar Transistor",
             strcat(MYKIT_BITMAP_DIR,"mykit_npn") );
```

Note Do not include the file extension (.bmp) on the bitmap name in **de_define_palette_group()**. This will cause the function to fail on some unix systems.

3. Save the file as $HOME/my_design_kit/de/ael/palette.ael
4. In order for ADS to read the new *palette.ael* file, the boot file must be modified to load the *palette.ael* file. Add the following line to my_design_kit/de/ael/boot.ael now:

   ```
   load(strcat(MYKIT_DE_AEL_DIR,"palette"), "CmdOp");
   ```

Note The **load()** function does not need the .ael extension on the filename it is loading. If the file *palette.atf* is found with a newer time stamp than *palette.ael*, the *palette.ael* file will not be recompiled. If *palette.ael* is newer, it will be recompiled into a new *palette.atf* and then loaded.

For more information about the palette function, refer to "Component Palette" on page 4-16.

At this time, you can test that the palette is loaded correctly. The files shown in Table 3-5 should now be saved in their respective directories under the design kit top level directory.

Table 3-5. Tutorial Directory and File Locations

Directory	File(s)
design_kit	ads.lib (plus a copy of it in $HOME/hpeesof/design_kit/ads.lib)
de/ael	boot.ael palette.ael
circuit/symbols	SYM_mykit_npn.dsn
circuit/ael	mykit_item.ael
doc	about.txt
circuit/bitmaps/pc	mykit_npn.bmp
circuit/bitmaps/unix	mykit_npn.bmp

To test your changes:

1. Restart ADS to load the design kit.
2. Open the *my_kit_prj* project in ADS.
3. From the schematic window, select the new palette called MY_DESIGN_KIT from the *Component Palette List*.
4. From the palette list on the left side panel of the schematic window, select the component from MY_DESIGN_KIT and drag it onto the schematic page.

Adding a Netlist Include Component

The tutorial sample design kit will get its model data from a netlist file with a model card that would typically be translated from another simulator such as HSpice, with the ADS Netlist Translator.

For the ADS simulator to find an externally referenced file, a design kit must contain a netlist include component. A built-in component called *NetlistInclude* is provided on the *Data Items* palette. On this component, you must browse to or manually enter the name of the included file. The built-in *NetlistInclude* component is handy for testing while building a design kit, but customers who use a design kit should not have to know where the model files are stored or be bothered with manually entering the file name. The tutorial will guide you through creating a custom include component where the file name is defined automatically for the design kit user.

Advanced Feature Tip If you place the *NetlistInclude* component and double click to view its parameters, select the *IncludeFiles* parameter and notice the field on the right side of the dialog called *Section*. This is one implementation of selecting a corner case as defined in a section of a model file. In your design kit, you will want to present the user with the list of available corner case labels, as opposed to making them fill in a blank. This is done by the definition of forms and formsets, as described in "Forms and Formsets" on page 4-11. An example of a component using this capability is also include in "Example Process Component with Forms and Formsets" on page 4-25.

Creating a Netlist Include Component Symbol

Create a netlist include component symbol, using the same process as described in "Creating Component Symbols" on page 3-7.

1. Open the *my_kit_prj* project in ADS if it is not already open.
2. In the schematic window, choose **View > Create/Edit Schematic Symbol**. The *Symbol Generator* dialog box appears.
3. In the Symbol Generator dialog box, select the *Copy/Modify* tab.
4. Set the Symbol Category to *Data Items*.
5. Click the *Netlist* symbol so the NetlistInclude appears in the Symbol Name field, then click **OK**.
6. Once you have copied the built-in symbol, you can edit the symbol.
7. It is beneficial for the words on the symbol to identify the design kit. Change the words *Netlist Include* to *My Kit Include*, as shown in Figure 3-2.

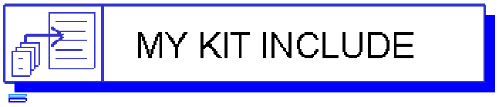


Figure 3-2. My Kit Include Component Symbol

For more information on editing symbol text, refer to "*Editing Existing Text and Text Attributes*" in Chapter 6 of the ADS "*User's Guide*".

8. Choose **File > Save Design As** and save the symbol file as a new symbol called, *SYM_mykit_include.dsn*.
9. Copy the saved symbol file, *SYM_mykit_include.dsn*, from the $HOME/my_kit_prj/networks directory to the $HOME/my_design_kit/circuit/symbols directory.

Creating the Netlist Include Component Bitmap

A bitmap must also be created for the new *mykit_include* component.

To create bitmaps for the include component, use the same general process as described in "Creating a Component Palette and Bitmaps" on page 3-14:

1. Create a bitmap that looks like the one in Figure 3-3. The *NetlistInclude.bmp* can be used to create your bitmap and is located under $HPEESOF_DIR/circuit/bitmaps.

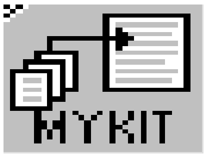


Figure 3-3. The *mykit_include.bmp* Bitmap

2. Save the UNIX and PC versions of the bitmap to the $HOME/my_design_kit/circuit/bitmaps/unix (and pc) directories. The bitmap files should be named *mykit_include.bmp*.
3. Modify the palette file (*palette.ael*) to add the new component to the palette as shown in Table 3-6.

Table 3-6. The Modified *palette.ael* File

```
de_define_palette_group(SCHEM_WIN, "analogRF_net",
        "MYKIT Components", "MYKIT Components", 0,
        "mykit_npn", "MYKIT NPN Bipolar Transistor",
                     strcat(MYKIT_BITMAP_DIR,"mykit_npn"),
        "mykit_include", "MYKIT Netlist Include",
                     strcat(MYKIT_BITMAP_DIR,"mykit_include") );
```

In a palette with a long list of components, make sure that a custom include or process component is placed at the top of the palette. This is a visual reminder to users that this component must be present in a schematic in order to simulate a circuit that uses any of the other components on that palette.

Modifying the Item Definition File

1. Open the $HOME/my_design_kit/circuit/ael/mykit_item.ael file.
2. Append the netlist callback and item definition for the include component to your existing item definition file (*mykit_item.ael*). The **create_item()** function call that you add should look like Table 3-7.

Table 3-7. The **create_item()** Function Call

```
set_simulator_type(1);

defun mykit_include_netlist_cb (cbP, clientData, callData)
{
    decl fileName="", netlistString="";
    fileName = strcat(MYKIT_CIRCUIT_MODEL_DIR, "mykit_models.net");
    netlistString=strcat(netlistString, "#include \"",
fileName,"\"\n");
    return(netlistString);
}
create_item("mykit_include",                              // name
            "MYKIT Netlist Include",                      // label
            "mykit_include",                              // prefix
            ITEM_UNIQUE|ITEM_NOT_NETLIST_IF_SUB,          // attribute
            -1,                                           // priority
            NULL,                                         // iconName
            standard_dialog,                              // dialogName
            NULL,                                         // dialogData
            ComponentNetlistFmt,                          // netlistFormat
            "mykit_include",                              // netlistData
            ComponentAnnotFmt,                            // displayFormat
            "SYM_mykit_include",                          // symbolName
            no_artwork,                                   // artworkType
            NULL,                                         // artworkData
            0,                                            // extraAttrib
            list (dm_create_cb (ITEM_NETLIST_CB,
"mykit_include_netlist_cb", NULL, TRUE)));
```

This item does not contain any parameters, but it does contain a netlist callback on the last line. This is provided so that when the component is netlisted for the simulator, the netlisting code can determine the location of the model file to include. Using path information saved in *boot.ael*, the callback generates a **#include** statement that points to the file *mykit_models.net* with a full path.

Notice that the netlist callback function is defined before the **create_item()** function. This is required if the .ael file will be compiled into an .idf file for demand-loaded components.

For more information on netlist callbacks, refer to "Netlist Callbacks" on page 6-14.

Adding a Model File

For this tutorial example, a simple NPN model with a few default values will be added to the design kit.

To add the my_design_kit/circuit/models/mykit_models.net file:

1. Open a text editor.
2. Copy the text below into a file.

```
model MYKIT_NPN_MODEL BJT NPN=1 PNP=0 Lateral=0 RbModel=0 Approxqb=1
```

3. Save the model file as:

   $HOME/my_design_kit/circuit/models/mykit_models.net

Your design kit should now contain the files shown in Table 3-8.

Table 3-8. Tutorial Directory and File Locations

Directory	File(s)
design_kit	ads.lib (plus a copy of it in $HOME/hpeesof/design_kit/ads.lib)
de/ael	boot.ael palette.ael
circuit/symbols	SYM_mykit_npn.dsn SYM_mykit_include.dsn
circuit/ael	mykit_item.ael
doc	about.txt
circuit/bitmaps/pc	mykit_npn.bmp mykit_include.bmp
circuit/bitmaps/unix	mykit_npn.bmp mykit_include.bmp
circuit/models	mykit_models.net

Creating an Example Design Using your Design Kit

Now, your design kit is ready for use in designing a circuit.

To use your new design kit:

1. Restart ADS to make sure all the information is loaded.
2. Start a new project and open a schematic window. Name the project *bjt_dc_prj*.
3. Using the BJT and Include components from your design kit, enter the schematic shown in Figure 3-4.

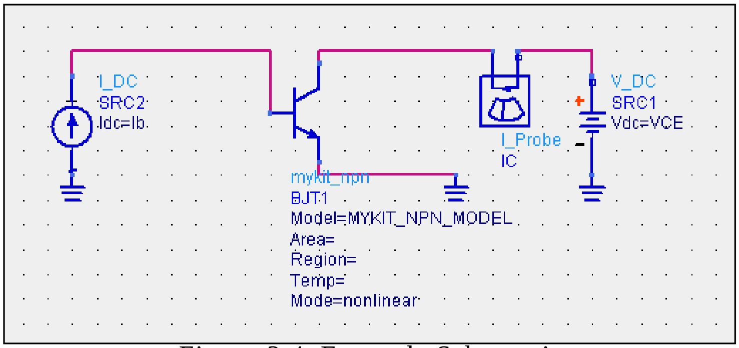
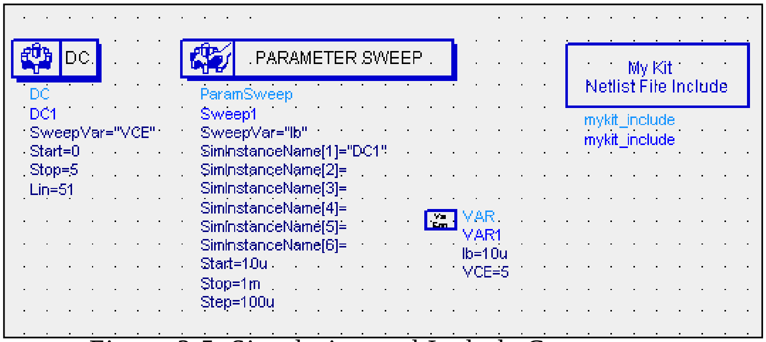


Figure 3-4. Example Schematic

4. Set the model parameter on the BJT to MYKIT_NPN_MODEL. This is the same name as the model in the model file created in "Adding a Model File" on page 3-22.
5. Insert the simulation components and include component shown in Figure 3-5 onto the schematic started in step 3. Figure 3-4 and Figure 3-5 will be combined into one schematic.


DC
DC1
SweepVar="VCE"
Start=0
Stop=5
Lin=51

ParamSweep
Sweep1
SweepVar="Ib"
SimInstanceName[1]="DC1"
SimInstanceName[2]=
SimInstanceName[3]=
SimInstanceName[4]=
SimInstanceName[5]=
SimInstanceName[6]=
Start=10u
Stop=1m
Step=100u

VAR
VAR1
Ib=10u
VCE=5

mykit_include
mykit_include

Figure 3-5. Simulation and Include Components

6. Simulate the circuit and plot IC.i vs. VCE. Your plot should look like the plot shown in Figure 3-6.

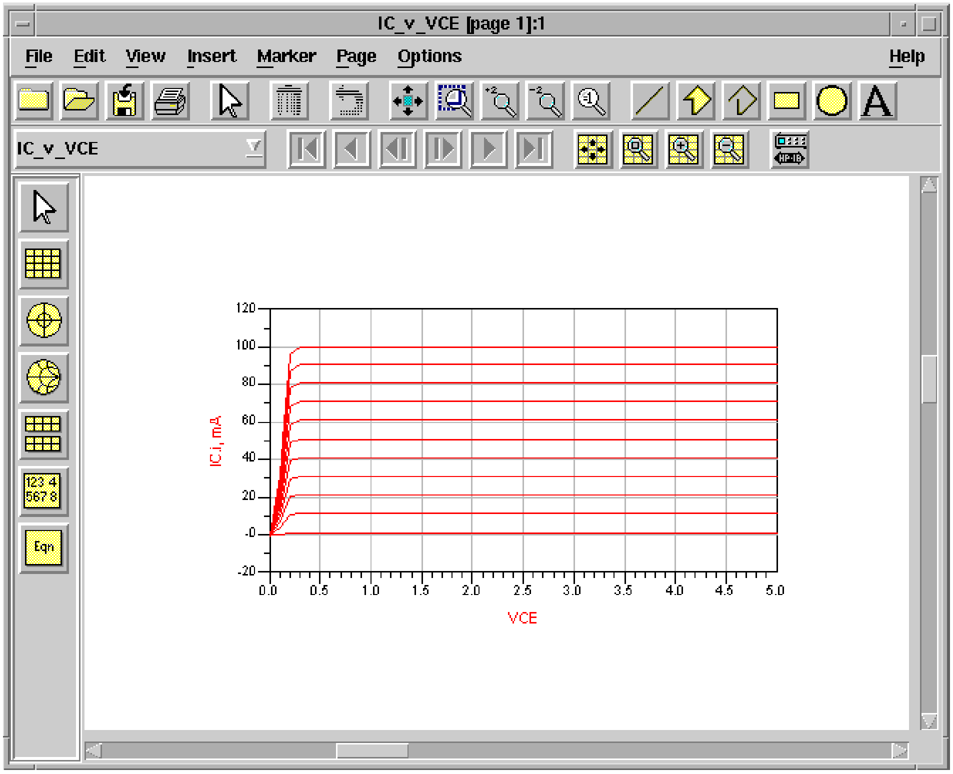


Figure 3-6. IC.i vs. VCE Simulation Results

7. Save the design and the data display. Call the files *bjt_dc.dsn* and *bjt_dc.dds*.
8. Using the **File > Archive** menu pick from the ADS *Main* window, archive the project. Save the archived project as *bjt_dc_prj.zap* in the my_design_kit/examples directory.

This completes the first part of the tutorial. You have now created a simple design kit. You could, at this point, refer to "Packaging for Distribution" on page 5-2 to prepare your design kit for distribution to an end user.

The example design kit that is shipped with the ADS 2001 Add-on release software is included in an archived copy of the project saved at this state. The sample design kit is located in $HPEESOF_DIR/examples/DesignKit/bjt_dc_prj/design_kit. From the ADS unarchive dialog box, you can browse to the examples directory of the shipped design kit and unarchive it anywhere on your system to compare it to the example you have just created.

The last part of the tutorial will cover the following topics:

- "Adding Components to the Library Browser" on page 3-26
- "Adding Demand Loaded Components" on page 3-29
- "Using a Subcircuit Model" on page 3-31
- "Adding a Resistor with SDD Subcircuit Model" on page 3-36
- "Accessing the Supplied Sample Kit" on page 3-46

Adding Components to the Library Browser

This is an optional step. If you choose to access your components from the component palette only, you may skip this step. The many advantages of the library browser, especially when using the control and records files, are outlined in "Component Palette vs. Library Browser" on page 4-15. Information is also provided in to help you decide if you should use the library browser in its simple form, complex form, or not at all.

Using the library_group() Function

The simplest way to setup design kit components to be visible in the library browser is by using the **library_group()** function. This is added to the *palette.ael* file.

To setup your tutorial components to be visible in the library browser:

1. In a text editor, open the $HOME/my_design_kit/de/ael/palette.ael file.
2. Add the following line to the end of the file.

   ```
   library_group("MYKIT", "MYKIT Components", "mykit_npn",
   "mykit_include");
   ```

   Note that the names of the components that are listed here must match the names in the item definition file *mykit_items.ael*. For more details on the **library_group()** function, refer to "Library Browser" on page 4-18, as well as Chapter 15 of the "*AEL*" documentation.
3. Save the *palette.ael* file.
4. Restart ADS and open the library browser from the **Insert > Component > Component Library** menu pick in the schematic window.
5. Verify that the library is listed, that the components are selectable, and that you can place the components in a schematic.

Setting Up a Control and Records Files

The second method for defining libraries for the library browser consists of setting up a control file and a set of one or more records files. For more information on the details of the file formats, refer to "Library Browser" on page 4-18.

To setup the control file, my_design_kit/circuit/records/mykit.ctl:

1. In a text editor, open the $HOME/my_design_kit/de/ael/palette.ael file that you edited in "Using the library_group() Function" on page 3-26.
2. Comment out the **library_group()** function call that you entered in the palette.ael file in the previous section of the tutorial.
3. Open a text editor and create the file shown in Table 3-9.

Table 3-9. The Control File (*mykit.ctl)*

```
<?xml version="1.0" ?>
<LIBRARIES>
  <LIBRARY>
    <NAME>MYKIT from control file</NAME>
    <CATEGORY>DL</CATEGORY>
    <RECORD_FILES>mykit.rec</RECORD_FILES>
  </LIBRARY>
</LIBRARIES>
```

4. Save the new file as *mykit.ctl* in the $HOME/my_design_kit/circuit/records directory.
5. Continue to the next section to write the records file.

To create the records file, which is referred to in the control file in Table 3-9:

1. In a text editor, create the file my_design_kit/circuit/records/mykit.rec, as shown in Table 3-10.

Table 3-10. The Records File (*mykit.rec)*

```
<?xml version="1.0" ?>
<COMPONENTS>
  <COMPONENT>
    <NAME>mykit_npn</NAME>
    <DESCRIPTION>MYKIT NPN Bipolar Transistor</DESCRIPTION>
    <LIBRARY>MYKIT</LIBRARY>
    <PLACEMENT>NOLAYOUT</PLACEMENT>
  </COMPONENT>
  <COMPONENT>
    <NAME>mykit_include</NAME>
    <DESCRIPTION>MYKIT Netlist Include</DESCRIPTION>
    <LIBRARY>MYKIT</LIBRARY>
    <PLACEMENT>NOLAYOUT</PLACEMENT>
  </COMPONENT>
</COMPONENTS>
```

2. Save the new file as *mykit.rec*, in the $HOME/my_design_kit/circuit/records directory.
3. After saving both the control and records files, restart ADS. The *mykit.ctl* and *mykit.rec* files will be located automatically by the software when the design kit is loaded.
4. Open a project and a schematic window.
5. Access the library browser and verify that the library is now called *"MYKIT from control file"*. If it is displayed as expected, edit the control file to remove the words *"from control file"*. This text was inserted temporarily to ensure that the browser entries actually came from the control file and not from the **library_group()** function as defined in "Using the library_group() Function" on page 3-26.

Note The format of the control and records files is very strict. Each space, bracket and slash is very important. If you do not see your library in the library browser, check your files very carefully against the files in Table 3-9 and Table 3-10. Correct any errors that you may find and restart ADS.

To make it easier to see your library, collapse all libraries from the toolbar by clicking the *Collapse Libraries* icon in the *Component Library/Schematic* dialog box.

Adding Demand Loaded Components

There is one more optional step in the tutorial related to setting up components. You can specify that components be loaded into ADS only when they are selected for placement. This is called *demand-loading*. As described in "Demand Loaded Components" on page 4-21, one more file can be added to the circuit/records directory. This file is called an *item definition file* (* .idf) and is a binary version of all the item definitions created in the *mykit_item.ael* file.

Note The example design kit shipped with ADS was saved after the previous section so it does not contain the .idf file described in this section. This is because this section replaces some of the files created in previous sections of the tutorial.

The advantages of using an item definition file are:

- Dynamic loading of components
- Faster loading speed
- Lower memory usage

When the item definition file (.idf) is used, components are loaded into a hash table and then loaded into ADS only when needed. For a large kit, there is a substantial loading time and memory usage savings.

The .idf file is created by a utility program called *hpedlibgen* that is shipped with ADS and is stored in the bin directory under $HPEESOF_DIR (the ADS installation directory).

Note $HPEESOF_DIR/bin must be in your path to use this utility, however, it should already be in your path if you are running ADS.

For the next step in the tutorial,

1. Open a text editor and create a file in the circuit/records directory called *mykit.list*. This file will be used by *hpedlibgen* and must contain the name of all AEL files with item definitions in them, one filename per line in the list file. For our small kit, there is only one file. A larger kit may use one file per component.
2. The file *mykit.list* for the tutorial should contain the single line:

   ```
   ../ael/mykit_item.ael
   ```

   The section on "Demand Loaded Components" on page 4-21 describes how the **create_item()** function call must be the last function in the block of functions related to each component. The program *hpedlibgen* uses the end of this function as the key to start reading the next item. This tutorial has not set up any parameter callbacks or other items that would typically be stored with the item definition other than the single netlist callback for the include component.
3. Enter the command to create the .idf file now. The syntax for running the program for the tutorial is:

   ```
   hpedlibgen -list mykit.list -out mykit.idf
   ```

To test the file, follow the steps below to remove the other methods of defining components and palettes that have been described so far in the tutorial.

1. Move the file circuit/ael/mykit_item.ael away to a safe place outside of the design kit directory.
2. Edit the file de/ael/boot.ael and comment out the line to load mykit_item.ael as shown below.

   ```
   // load(strcat(MYKIT_CIRCUIT_AEL_DIR,"mykit_item"), "CmdOp");
   ```

   Note that comment characters in AEL are two forward slashes at the beginning of the line (//) or the traditional C programming slash and asterisk pair delimiting a complete block of code (/* this text is commented out */).
3. In *palette.ael*, make sure the **library_group()** function call is commented out.

Restart ADS to make sure the components are still available and can be placed from the palette and from the library browser. Their visibility on the palette or in the library browser is not an indication of success. The demand loading will take place when the items are selected and placed. This can be verified by checking the communication log between ADS and the library browser for a query of the form:

```
QUERY_ITEM_DEFINITION_RESPONSE dsn-type component-name library-name
library-title ATF-file file-date file-location
```

The communication log will be generated if ADS is started with the optional command line argument "*-d daemon.log*". On a PC, add this to the shortcut property "*Target*". The file daemon.log will be saved into the directory from which ADS is started.

Using a Subcircuit Model

The section on "Adding a Model File" on page 3-22 explains how to set up a component that refers to a model card in an included netlist file. It is also possible to use a subcircuit model in the same manner if a simple model card is not sufficient to represent the model.

As with a simple model card, the subcircuit model is typically translated from another simulator with the ADS SPICE or Spectre Netlist Translator, or it can be generated from IC-CAP.

The files or code fragments listed here can be added to the design kit created in the tutorial as an example of a subcircuit model for the BJT transistor.

The files that need to be modified or added are:

circuit/ael/mykit_item.ael	-	add the new item definition
circuit/bitmaps	-	add new bitmaps
circuit/models/mykit_models.net	-	add new model to the model file
circuit/records/mykit.rec file	-	add new component
circuit/records/mykit.idf file	-	recompile to add new component
circuit/symbols	-	no change, ok to reuse the same symbol
de/ael/palette.ael	-	add new component

The new content for each file is listed below. Refer to the earlier sections in the tutorial for a reminder of the steps required to update the sample kit.

The steps below provide a general outline of the process.

1. Add the new item definition to circuit/ael/mykit_item.ael as shown in Table 3-11.

Table 3-11. my_design_kit/circuit/ael/mykit_item.ael

```
set_simulator_type(1);

create_item( "mykit_npn_subckt",
             "MYKIT Nonlinear Bipolar Transistor Subckt, NPN",
             "BJT",
             NULL, NULL, NULL,
             standard_dialog, "",
             CmpModelNetlistFmt, "",
             ComponentAnnotFmt,
             "SYM_mykit_npn",
             no_artwork, NULL,
             ITEM_PRIMITIVE_EX,
create_parm("Model", "Model instance name", PARM_NOT_EDITED,
"StdFileFormSet",UNITLESS_UNIT,prm("StdForm","bjt_nhf"))
);
```

2. Add the new bitmaps (see Figure 3-7) to the pc and unix subdirectories.

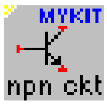


Figure 3-7. my_design_kit/circuit/bitmaps/mykit_npn_subckt.bmp

3. Add the new model to the model file circuit/models/mykit_models.net as shown in Table 3-12.

Table 3-12. my_design_kit/circuit/models/mykit_models.net

```
; subckt model for npn
define bjt_nhf (C B E )
L:LE E 4 L=1.015E-10
L:LB B 5 L=1.959E-10
C:CC C 0 C=2.505E-13
NPN:Q1 C 5 4 Area = 1

model NPN BJT NPN=yes \
Is = 1.467E-15 Bf = 221.5 Nf = 0.9915 Vaf = 44.63 Ikf = 100 \
Ise = 2.823E-13 Ne = 2.5 Br = 6.493 Nr = 0.9902 Var = 1.841 \
Ikr = 100 Isc = 5.706E-15 Nc = 1.171 Rb = 3.579 Irb = 1E-12 \
Rbm = 3.202 \
Re = 0.4211 \
Rc = 0.3492 \
Xtb = 0 \
Eg = 1.11 \
Xti = 3 \
Cje = 2.458E-12 \
Vje = 1.004 \
Mje = 0.502 \
Tf = 1.886E-11 \
Xtf = 13.97 \
Vtf = 0.2296 \
Itf = 2.225 \
Ptf = 30.72 \
Cjc = 1.685E-12 \
Vjc = 0.6296 \
Mjc = 0.3898 \
Xcjc = 0.3 \
Tr = 1E-09 \
Cjs = 9.985E-14 \
Vjs = 0.8137 \
Mjs = 0.3509 \
Fc = 0.9 \
Tnom = 27
end bjt_nhf
```

Note In the file above, mykit_models.net, the *define* keyword is used to define a subcircuit model. ADS does not support nested subcircuits, so another define cannot be included before the matching *end* keyword. A design kit user can inadvertently cause this to happen. See "Avoiding Illegal Nested Subcircuits" on page 3-35 for tips on how to prevent this from happening.

4. Add the new component to the records file circuit/records/mykit.rec as shown in Table 3-13.

Table 3-13. my_design_kit/circuit/records/mykit.rec

```
<COMPONENT>
 <NAME>mykit_npn_subckt</NAME>
 <DESCRIPTION>MYKIT NPN Bipolar Transistor Subckt</DESCRIPTION>
 <LIBRARY>MYKIT</LIBRARY>
 <PLACEMENT>NOLAYOUT</PLACEMENT>
</COMPONENT>
```

5. Recompile the item definition file to add the new components.

```
hpedlibgen -list mykit.list -out mykit.idf
```

6. Since you can re-use the same symbols, there is no need to change the circuit/symbols.
7. Add the new component to the de/ael/palette.ael file as shown in Table 3-14.

Table 3-14. my_design_kit/de/ael/palette.ael

```
de_define_palette_group(SCHEM_WIN, "analogRF_net", "MYKIT Components",
             "MYKIT Components", 0,
             "mykit_npn", "MYKIT NPN Bipolar Transistor",
                    strcat(MYKIT_BITMAP_DIR,"mykit_npn"),
             "mykit_npn_subckt", "MYKIT NPN Bipolar Transistor Subckt",
                    strcat(MYKIT_BITMAP_DIR,"mykit_npn_subckt"),
             "mykit_include", "MYKIT Netlist Include",
                    strcat(MYKIT_BITMAP_DIR,"mykit_include")
);
```

8. The new component is used the same way the old component was used. In the example project, delete the existing npn component and place an instance of the new subckt npn.

Avoiding Illegal Nested Subcircuits

ADS does not support nested subcircuits. Nested subcircuits will occur in a netlist when a hierarchical schematic has an include component, that includes a file with a subcircuit, which is used in any level of hierarchy other than the top level. To prevent this from occurring, use the ITEM_NOT_NETLIST_IF_SUB attribute code when defining the netlist include component. This attribute code is described in "Attribute Code Examples" on page 4-7. This will not prevent the user from placing the component in a lower level of hierarchy but it will prevent the component from being netlisted. Your design kit instructions should include a warning that include components only be placed in the top level of hierarchy. Otherwise your users will see an error message during simulation indicating that the system cannot find the models that are in the included netlist because the netlist will not have been included.

Adding a Resistor with SDD Subcircuit Model

This section describes another component that uses a subcircuit model in a netlist file. It is included in the tutorial to show a simple example of using a Symbolically Defined Device (SDD) to define arbitrary current/voltage relationships. It also shows the need for defining a noise voltage source for this *resistor*, which is calculated using the relationship Vn=4kTReff.

Two versions of the resistor model are included with different resistivities (ρ) to demonstrate the use of forms and selecting types from the edit component dialog when placing a component. An example of a circuit which contains this component is shown in Figure 3-8 and is also included in the *bjt_dc_prj* shipped with the sample design kit code.

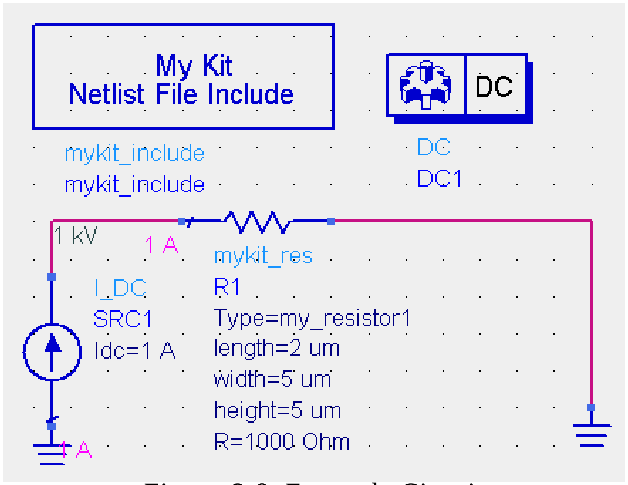


Figure 3-8. Example Circuit

In this example, a single current source is used to drive 1 amp through the resistor such that the value of the voltage across the resistor equals the resistance value specified on the component. Performing a simulation and then annotating the DC results (use the menu pick **Simulate > Annotate DC Solution**) will verify that the correct model is being selected for simulation: the annotated voltage should equal the resistance specified on the component.

You do not always need to create a custom symbol for each design kit component. Sometimes you can use a built-in ADS symbol. In this case, the item definition in *mykit_res.ael* uses the built-in ADS resistor symbol SYM_R.

The file or code fragments listed here can be added to the design kit created in the tutorial. These files are also included with the ADS software in the $HPEESOF_DIR/examples/DesignKit/bjt_dc_prj/design_kit directory. It is highly recommended that you copy the file *mykit_res.ael* from this location since copying it from the text in the documentation is highly prone to introducing errors due to wrapping of text.

The files to modify or add to the design kit are listed below. Refer to the tutorial for the actual steps for updating the sample kit.

circuit/ael/mykit_res.ael	-	save the new item definition
circuit/bitmaps	-	add the new bitmaps mykit_res.bmp
circuit/models/mykit_models.net	-	add the new model
de/ael/boot.ael	-	add code to load AEL file mykit_res.ael
de/ael/palette.ael	-	add new component to the palette
optional:		
circuit/records/mykit.rec	-	add new component for the library browser
circuit/records/mykit.list	-	add new item definition file mykit_res.ael for hpedlibgen
circuit/records/mykit.idf	-	recompile to add new component for demand loading

The steps below provide a general outline of the process.

1. Copy the new item definition for the resistor from the file *mykit_res.ael* in the $HPEESOF_DIR/examples/DesignKit/bjt_dc_prj/design_kit directory. This file is shown on the following pages.

```
/*-----------------------------------------------------------------------+/
```

```
   FILE          : mykit_res.ael
   COMMENTS      : Component definition :
                     [ global variables ]
                     [ forms and formsets ]
                     [ netlist callback function ]
                     [ parameter callback functions ]
                     item definition
/+---------------------------------------------------------------------------*/
/*--- utility function to check parameter value with respect to ranges ---*/
defun parm_checkRange
(
  name,
  actual,
  min,
  max,
  unitString
)
{
  // check within some relative margin to compensate for round-off error.

  decl scale;

  if (is_string(unitString) && strlen(unitString) > 0)
    scale = evaluate(sprintf("1 %s", unitString));
  else
  {
    scale = 1;
    unitString = "";
  }

  if (max > 0.0 && (actual-max)/max > 1.e-5)
  {
    warning("aelcmd", 16, "", fmt_tokens(list("The maximum", name,
                                           "is", max/scale, unitString, "."))
);
  }
  else if (min > 0.0 && (min-actual)/min > 1.e-5)
  {
    warning("aelcmd", 16, "", fmt_tokens(list("The minimum", name,
                                           "is", min/scale, unitString, "."))
);
  }
  return;
}
/*--- global variables ------------------------------------------------------*/
decl res_type_default = "form_res_type_myres1";
decl res_length_default = 3.0e-6;
decl res_width_default = 5.0e-6;
decl res_height_default = 5.0e-6;

// typ model parameters: list(typeLabel, list(rho), ...
decl res_modelList = list("my resistor 1 (length >= 2.0um)",
                                       list(12.5e-3),
                                  "my resistor 2 (length >= 1.0um)",
                                       list(8.5e-3)
);

// Width ranges: list(typeLabel, list(lmin, lmax), ...)
decl res_ranges = list("my resistor 1 (length >= 2.0um)",
                                    list(2.0e-6, 0.0),
                             "my resistor 2 (length >= 1.0um)",
                                    list(1.0e-6, 0.0)
                                    );

/*--- forms and formsets --------------------------------------------------*/
// Type
create_constant_form("form_res_type_myres1",
                     "my resistor 1 (length >= 2.0um)", 0, "myres1",
"my_resistor1");
create_constant_form("form_res_type_myres2",
                     "my resistor 2 (length >= 1.0um)", 0, "myres2",
"my_resistor2");

create_form_set("formset_res_type","form_res_type_myres1",
"form_res_type_myres2");

/*--- netlist callback function ------------------------------------------*/

/*--- default value parameter callback function --------------------------*/
defun res_parm_defaultValue_cb
(
  cbP,
  parmName,
  parmDefP
)
{
  decl parmH = NULL;

/*--- calculate default parameter value ---------------------------------*/
  if (!strcmp(parmName, "Type"))
  {
    parmH = prm(res_type_default);
  }
  else if (!strcmp(parmName, "length"))
  {
    parmH = prm("StdForm", sprintf("%g um", res_length_default/1.0e-6));
  }
  else if (!strcmp(parmName, "width"))
  {
    parmH = prm("StdForm", sprintf("%g um", res_width_default/1.0e-6));
  }
  else if (!strcmp(parmName, "height"))
  {
    parmH = prm("StdForm", sprintf("%g um", res_height_default/1.0e-6));
  }
  else if (!strcmp(parmName, "R"))
  {
    decl formDefP = dm_find_form_definition(res_type_default);
    decl formLabel = dm_get_form_definition_attribute(formDefP,
DM_FORM_LABEL);

    decl modelList = car(cdr(member(formLabel, res_modelList)));
    decl Rho = nth(0, modelList);

    decl R = Rho*res_length_default / res_width_default / res_height_default;

    R = round(R*1.0e2)*1.0e-2; // round to 0.01 Ohm
    parmH = prm("StdForm", sprintf("%g Ohm", R));
  }

  /*--- return to calling function ----------------------------------------*/
  return parmH;
}


/*--- modified value parameter callback function ------------------------*/
defun res_parm_modified_cb
(
  cbP,       // callback function pointer
  parmName,  // clientData
  itemInfoP  // itemInfo pointer
)
{
decl parmList = NULL;
  decl type, modelList, rangeList;
  decl Rho;
  decl L, W, H, R;

  /*--- calculate dependent parameter values ------------------------------*/
  type = pcb_get_form_value(itemInfoP, "Type");
  modelList = car(cdr(member(type, res_modelList)));
  rangeList = car(cdr(member(type, res_ranges)));
  if (listlen(modelList) == 1)
  {
    Rho = nth(0, modelList);

    if (!strcmp(parmName, "length") || !strcmp(parmName, "width")
                        || !strcmp(parmName, "height") || !strcmp(parmName,
"Type"))
    {
      // get mks value of independent parameter(s)
      L = pcb_get_mks(itemInfoP, "length");
      W = pcb_get_mks(itemInfoP, "width");
      H = pcb_get_mks(itemInfoP, "height");
      parm_checkRange("resistor length", L, nth(0, rangeList),
nth(1, rangeList), "um");

      R = Rho * L / W / H;
      R = round(R*1.0e2)*1.0e-2; // round to 0.01 Ohm

      // build return structure with dependent parameter(s)
      parmList = pcb_set_mks(parmList, "R", R);
    }
    else if (!strcmp(parmName, "R"))
    {
      // get mks value of independent parameter(s)
      R = pcb_get_mks(itemInfoP, "R");
      W = pcb_get_mks(itemInfoP, "width");
      H = pcb_get_mks(itemInfoP, "height");

      // calculate dependent parameter(s)
        L = R / Rho * W * H;
      L = round(L*1.0e8)*1.0e-8; // round to 0.01 um
        parm_checkRange("resistor length", L,
                                nth(0, rangeList), nth(1, rangeList), "um");

      // build return structure with dependent parameter(s)
      parmList = pcb_set_mks(parmList, "length", L);
    }
  }

  /*--- return to calling function ----------------------------------------*/
  return parmList;
}


/*--- item definition ---------------------------------------------------*/
create_item(
  "mykit_res",                                       // name
  "An ideal resistor",                               // description label
  "R",                                               // prefix
  0,                                                 // attributes
  NULL,                                        // priority
  "MY_RES",                                    // iconName
standard_dialog,                               // dialogName
  NULL,                                        // dialogData
  "%0b'%p'%1e:%t %# %44?0%:%31?%C%:_net%c%;%;%e %1b%r%8?%29?%:%30?%p
%:%k%?[%1i]%;=%p %;%;%;%e%e",
//  CmpModelNetlistFmt,                           // netlist format string
  NULL,                                           // netlist data
  ComponentAnnotFmt,                              // display format string
  "SYM_R",                                        // symbol name
  no_artwork,                                     // artwork type
  NULL,                                           // artwork data
  ITEM_PRIMITIVE_EX                               // extra attributes
  ,create_parm(                                   // parameter
    "Type",                                       // name
    "Resistor type",                              // label
    PARM_DISCRETE_VALUE,                          // attrib
    "formset_res_type",                           // formSet
    UNITLESS_UNIT,                                // unit code
    prm(res_type_default)                         // default value
    ,list(dm_create_cb(PARM_DEFAULT_VALUE_CB,  // parameter default value
                                                  // callback function
                       "res_parm_defaultValue_cb", // function name
                       "Type",                    // clientData
                       TRUE),                     // callback enableFlag
dm_create_cb(PARM_MODIFIED_CB,                // parameter modified value
                                                  // callback function
                       "res_parm_modified_cb",    // function name
                       "Type",                    // clientData
                       TRUE)                  // parameter modified value
                                                  // callback function
          )
    )
  ,create_parm(                                   // parameter
    "length",                                     // name
    "length, R=f(length, width, height)",         // label
    PARM_OPTIMIZABLE|PARM_STATISTICAL,            // attrib
    "StdFileFormSet",                             // formSet
    LENGTH_UNIT,                                  // unit code
    prm("StdForm", "3 um")                        // default value
    ,list(dm_create_cb(PARM_DEFAULT_VALUE_CB,    // parameter default value
                                                  // callback function
                       "res_parm_defaultValue_cb", // function name
                       "length",                  // clientData
                       TRUE),                     // callback enableFlag
         dm_create_cb(PARM_MODIFIED_CB,          // parameter modified value
                                                  // callback function
                       "res_parm_modified_cb",    // function name
"length",                                              // clientData
                      TRUE)                          // parameter modified value
                                                       // callback function
          )
    )
 ,create_parm(                                          // parameter
    "width",                                            // name
    "width, R=f(length,width,height)",                  // label
    PARM_OPTIMIZABLE|PARM_STATISTICAL,                  // attrib
    "StdFileFormSet",                                   // formSet
    LENGTH_UNIT,                                        // unit code
    prm("StdForm", "5 um")                              // default value
    ,list(dm_create_cb(PARM_DEFAULT_VALUE_CB,   // parameter default value
                                                        // callback function
                       "res_parm_defaultValue_cb",  // function name
                       "width",                         // clientData
                       TRUE),                           // callback enableFlag
          dm_create_cb(PARM_MODIFIED_CB,        // parameter modified
                                                      // value callback function
                       "res_parm_modified_cb",      // function name
                       "width",                         // clientData
                       TRUE)                    // parameter modified
                                                      // value callback function
)
    )
 ,create_parm(                                          // parameter
    "height",                                           // name
    "height, R=f(length, width, height)",               // label
    PARM_OPTIMIZABLE|PARM_STATISTICAL,                  // attrib
    "StdFileFormSet",                                   // formSet
    LENGTH_UNIT,                                        // unit code
    prm("StdForm", "5 um")                              // default value
    ,list(dm_create_cb(PARM_DEFAULT_VALUE_CB,   // parameter default
                                                      // value callback function
                       "res_parm_defaultValue_cb",  // function name
                       "height",                        // clientData
                       TRUE),                           // callback enableFlag
          dm_create_cb(PARM_MODIFIED_CB,        // parameter modified
                                                      // value callback function
                       "res_parm_modified_cb",      // function name
                       "height",                        // clientData
                       TRUE)                    // parameter modified
                                                      // value callback function
          )
    )
  ,create_parm(                                         // parameter
)
    )
```

```
,create_parm(                                               // parameter
  "R",                                                      // name
  "R, length=f(R, width, height)",                          // label
  PARM_REAL|PARM_NOT_NETLISTED,                             // attrib
  "StdFileFormSet",                                         // formSet
  RESISTANCE_UNIT,                                          // unit code
  prm("StdForm", "")                                        // default value
  ,list(dm_create_cb(PARM_DEFAULT_VALUE_CB,             // parameter default
                                                           // value callback function
                     "res_parm_defaultValue_cb",            // function name
                     "R",                                   // clientData
                     TRUE),                                 // callback enableFlag
        dm_create_cb(PARM_MODIFIED_CB,                  // parameter modified
                                                           // value callback function
                     "res_parm_modified_cb",                // function name
                     "R",                                   // clientData
                     TRUE)                              // parameter modified
                                                           // value callback function
        )
  )
);
```

2. Add the new bitmaps (see Figure 3-9) to the pc and unix subdirectories.

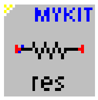


Figure 3-9. mykit_res.bmp

3. Add the new model to the model file as shown in Table 3-15.

Table 3-15. my_design_kit/circuit/models/mykit_models.net

```
; subckt models for resistor with SDD

define myres1 ( n1 n2)
parameters  length= width= height=

l=length
w=width
h=height
rho=12.5e-3

reff=rho*l/h/w
vn=sqrt(4*boltzmann*(273+temp)*reff)
V_Source:vn  n1 _net1 V_Noise=vn SaveCurrent=0

SDD:r1 _net1 n2 n2 n1 I[2,0]=0 I[1,0]=(_v1)/(rho*l/h/w)
end myres1

define myres2 ( n1 n2)
parameters  length= width= height=

l=length
w=width
h=height
rho=8.5e-3

reff=rho*l/h/w
vn=sqrt(4*boltzmann*(273+temp)*reff)
V_Source:vn  n1 _net1 V_Noise=vn SaveCurrent=0

SDD:r1 _net1 n2 n2 n1 I[2,0]=0 I[1,0]=(_v1)/(rho*l/h/w)
end myres2
```

4. In my_design_kit/de/ael/boot.ael, add the code to load the AEL file *mykit_res.ael*.

```
load(strcat(MYKIT_CIRCUIT_AEL_DIR, "mykit_res"), "CmdOp");
```

5. Add the new component to the palette.ael file as shown in Table 3-16.

Table 3-16. my_design_kit/de/ael/palette.ael

```
de_define_palette_group(SCHEM_WIN, "analogRF_net", "MYKIT
Components",
        "MYKIT Components", 0,
        "mykit_npn", "MYKIT NPN Bipolar Transistor",
                strcat(MYKIT_BITMAP_DIR,"mykit_npn"),
        "mykit_res", "MYKIT Resistor with SDD",
                strcat(MYKIT_BITMAP_DIR,"mykit_res"),
        "mykit_include", "MYKIT Netlist Include",
                strcat(MYKIT_BITMAP_DIR,"mykit_include") );

library_group("MYKIT", "MYKIT Components", "mykit_npn", "mykit_res",
"mykit_include");
```

6. Add the new component to the circuit records files as shown in Table 3-17.

Table 3-17. my_design_kit/circuit/records/mykit_res.rec

```
<COMPONENT>
 <NAME>mykit_res</NAME>
 <DESCRIPTION>MYKIT Resistor with SDD</DESCRIPTION>
 <LIBRARY>MYKIT</LIBRARY>
 <PLACEMENT>NOLAYOUT</PLACEMENT>
</COMPONENT>
```

7. Add mykit_res.ael to the temporary file mykit.list. Rerun hpedlibgen to recompile the item definition file to add the new components.

```
hpedlibgen -list mykit.list -out mykit.idf
```

8. Since you can re-use the same symbols, there is no need to change the circuit/symbols.

Accessing the Supplied Sample Kit

A copy of the completed tutorial design kit is available with the ADS design kit software. This can be used to compare the files as you build them in the tutorial. However, since the tutorial builds the files slowly, it is recommended that you go through the process as described in the tutorial.

The sample design kit is located in:

$HPEESOF_DIR/examples/DesignKit/bjt_dc_prj/design_kit

Chapter 4: Basic Parts of an ADS Design Kit

This chapter details the basic parts of an ADS design kit. Chapter 3, ADS Design Kit Tutorial guided you step by step through building a design kit with each of the parts described in this chapter. As described in Chapter 2, Understanding the ADS Design Kit File Structure, the basic parts of a design kit are those that define the components, the simulation data, as well as a simple file to identify the design kit to ADS, and the AEL files required to load the design kit.

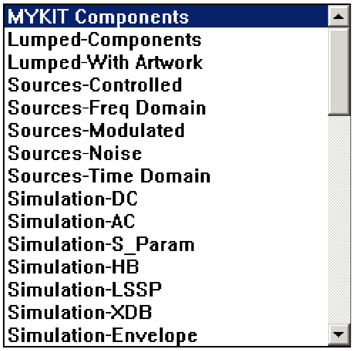
As you build your design kit, keep in mind that if your kit includes translated models, verification of the translated models is an essential part of design kit creation and may take more time than building the kit itself. For more information on the verification process, refer to "Verifying a Design Kit" on page 5-1.

Design Kit Name

Each design kit must have a unique name and the name cannot include spaces. Since an ADS user may have multiple design kits loaded, from different sources, the design kit name should be informative. It should contain the name of the company or foundry as well as the name and/or version of the process.

The name of the design kit is only used in a couple places. It is the name of the directory that all the other subdirectories are stored under, as illustrated in "Understanding the Directory Contents" on page 2-8. The design kit name is entered into the template *ads.lib* file and from there it becomes a global AEL variable whose value is set to the path to the design kit. The name is also visible in the dialogs for end users. These dialogs are used to load and set up the design kits that are accessible in ADS. For more information on the user interface, refer to the ADS "*Design Kit User's Guide*".

A design kit name may also be shown on the component palette, if a palette is provided, but this is not an automatic correlation. The palette title is set in a different location than the design kit name. A design kit may contain multiple palettes. Each palette should contain the name of the design kit in addition to whatever other information is appropriate and palette titles can have spaces in them. The palette is the location where the user will most often see the name of the design kit because the palette is visible at all times in the schematic window. The list of default system palettes, plus the example palette from the design kit tutorial, are shown in Figure 4-1.



MYKIT Components
Lumped-Components
Lumped-With Artwork
Sources-Controlled
Sources-Freq Domain
Sources-Modulated
Sources-Noise
Sources-Time Domain
Simulation-DC
Simulation-AC
Simulation-S_Param
Simulation-HB
Simulation-LSSP
Simulation-XDB
Simulation-Envelope

Figure 4-1. Design Kit Tutorial Palette Name

Components in a Design Kit

Defining a component in ADS requires entries in a number of files. Information must be provided to control the behavior of the component when used in a schematic or layout or when processed by the simulator. The component definition AEL file contains much of this information.

Components can be presented to the user in an ADS Schematic window palette or in the library browser or both. Symbols are required for both of these methods. Bitmaps are only required for the component palette. Additionally, you can set your design kit components up to be loaded on demand. For kits with large numbers of components, this reduces the impact on ADS startup time.

The items that must be defined for each component in a design kit are explained in the following sections:

- "Component Name" on page 4-2
- "Item Definition" on page 4-4
- "Schematic Symbol" on page 4-12
- "Component Palette vs. Library Browser" on page 4-15

Component Name

Component names are a unique identifier of each component loaded in ADS at any given time. The name is visible in the balloon that is displayed when you hold your cursor over a palette tile. It is also visible on the ADS schematic after you have placed your component. Eventually, the component name is passed to the simulator in the netlist.

Since the component name is a unique identifier, each component name must be *completely unique in the ADS system*, not just within a library. If another design kit uses the same name, the one that is loaded last will overwrite any that were loaded before it.

You can test a new component name by entering it in the *Component History* field as shown in Figure 4-2. The system is case-sensitive so names must be capitalized exactly as shown.

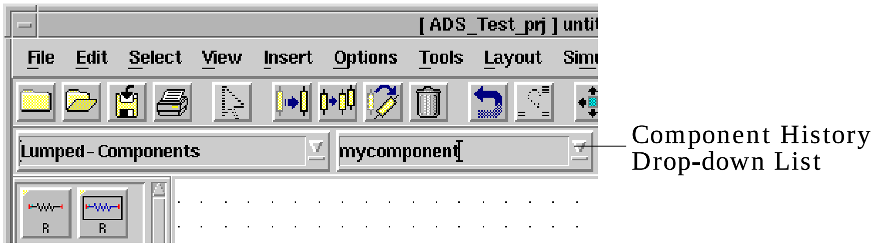


Figure 4-2. ADS Component History

If you receive the error *Failed to locate the component definition* after entering your new component name, this is an indication that the name is not currently being used in ADS. To ensure that your component names are completely unique, you should prefix each component name with an identifier which links it back to the design kit. For example, the name of an RFIC foundry component would include:

- The name of the foundry
- The specific foundry process
- The common device type identifier

Separate the words in your component name with an underscore to make it more readable.

Example:

<foundry>_<process>_<device type>

Another advantage of prefixing each component with a string such as the design kit name is that sometimes a design that was built with design kit components will get sent to someone who does not have that particular design kit loaded. In this case, when the design is opened, a generic message indicating that the component was not found is all that the user would see. Then, when they look at the schematic, there will be a skeletal symbol with only pins and the component name. Seeing the component name will help them determine the library that the component came from.

Note A component name in ADS cannot contain spaces.

Using Valid ADS Characters

The legal character set for Advanced Design System names is; alphanumeric _+ - = ^ ' @ # & $ %. Note that other than alphanumeric and underscore characters, all other legal characters require special handling in ADS. It is highly recommended that you use only alphanumeric and underscore characters in any part of your design kit.

Reserved Words

There is also a list of reserved words in ADS which cannot be used for component names. For more information on reserved words in ADS, refer to the section on "*Reserved Words*" in "*Appendix E*" of the "*RFIC Dynamic Link Library Guide*".

Item Definition

All components that exist in ADS are added by calling an AEL function called **create_item()**. Each component can have a number of parameters, which are added to the system by calls to the function **create_parm()**. The combination of these calls is referred to as the *item definition*. A user interface is provided to set up some of this information, but it also usually requires some additional manual editing. The tutorial gave an example of this process.

In addition to parameters and their default values, the item definition in ADS contains information about a component that is necessary for display on the schematic. There is also information included that is needed for simulation. The following section describes the syntax and the arguments of the **create_item()** function that are important for design kits. Additional information on the **create_item()** command is available in Chapter 15 of the *"AEL"* documentation.

The syntax for the **create_item()** command is shown here. The square brackets indicate optional parameters, but are not entered in the actual function call.

```
create_item(name, desc, prefix, attrib, priority, iconName, dialogCode,
            dialogData, netlistFormat, netlistData, displayFormat,
            symbolName,artworkType, artworkData [, extraAttribute, cbList,
            parameterN]);
```

An example similar to that in the tutorial is repeated here for reference. The callback and **create_parm()** information is incomplete but is sufficient for demonstration purposes. The parameter descriptions are included in Table 4-1.

```
create_item("mykit_npn", "MYKIT Nonlinear Bipolar Transistor", "BJT", NULL,
            NULL, NULL, standard_dialog, CmpModelNetlistFmt,
            ComponentAnnotFmt, SYM_mykit_npn, no_artwork, NULL,
            ITEM_PRIMITIVE_EX, list(callback1, callback2, etc),
            create_parm("parm1",...), create_parm("parm2",...));
```

Table 4-1. The **create_item()** Parameter Descriptions

Argument	**Description**
name	Name of the component. It must be completely unique in the system. For design kits, it should include a company and/or process reference.
desc	Description of the component. 80 character limit. This shows up in the edit component dialog as well as in the balloon help when the cursor is positioned over the component in the palette.
prefix	The prefix is typically 1-3 characters but can be longer. It is used to create a unique ID when the component is placed in a schematic.
attrib	This is a special attribute code. The AEL manual contains a list of codes in Table 15-1. Each code has a numerical value assigned to it. Use of these codes is explained below. A value of 0 or NULL indicates that no special codes are set.
priority	Not used for design kit components - set it to NULL.
iconName	Not used for design kit components - set it to NULL.
dialogCode	This tells the system which dialog to open for editing of parameters and other component attributes. It is usually set to standard_dialog, which is a constant value. A constant value is a special value that the system knows about. It is not a string. Do not put it in quotes.
dialogData	Not used for design kit components - set it to "".

Table 4-1. The **create_item()** Parameter Descriptions

Argument	Description
netlistFormat	This tells the system what format to use when the component is output to a netlist. Two constant values are defined that should be sufficient for most design kit components. These are not strings so do not put them in quotes. These netlist format values are: ComponentNetlistFmt - output all parameters as name=value. Use if component has no model reference. CmpModelNetlistFmt - output first parameter as Model=value and all others as name=value. Use if component has a model reference, which must be the first parameter. Other pre-defined formats can be seen in the file circuit/ael/gemini.ael in the ADS installation. If the pre-defined formats are not sufficient, the netlistFormat can be customized by use of a special notation described in "Format Strings" in the AEL manual. Since this is a complex topic, it is beyond the scope of this manual to provide comprehensive coverage of it.
netlistData	Not used for design kit components - set it to "".
displayFormat	This tells the system how to display the component annotation on the schematic. Component annotation consists of the component name and unique ID. For design kit components, use the value ComponentAnnotFmt. This is a constant known to the system, not a string. Do not put quotes around it.
symbolName	This is the name of the symbol that was created for the component. It is the name of the design file containing the symbol graphics, without the .dsn extension.
artworkType	This tells the system what type of artwork is associated with the component. The following values are used. The equivalent constant value in parentheses can also be used. 0 = no artwork (no_artwork) 1 = fixed artwork. Artwork that is not dependent on physical parameters and is stored in a .dsn file. (fixed_artwork) 2 = ael generated artwork. Artwork that is dependent on varying physical parameters and must be generated by running an AEL macro. (macro_artwork) 3 = synchronized. Used for a subnetwork. The artwork geometry is contained in the same design file as the schematic information.

Table 4-1. The **create_item()** Parameter Descriptions

Argument	**Description**
artworkData	If artworkType = 0 or 3, set this to NULL. If artworkType = 1, this string is set to the name of the design file containing the artwork. If artworkType = 2, this string is set to the name of the AEL function that will be executed to draw the artwork.
extraAttribute	This is another attribute code. The AEL manual contains a list of these codes in Table 15-2. Each code has a numerical value assigned to it. Use of these codes is explained below. A value of 0 or NULL indicates that no special codes are set.
cbList	List of netlist callbacks. See Chapter 6, Netlist Callbacks for details.
parameterN	All parameters of the component, each defined by a separate call to create_parm(). These are not presented in the AEL list() syntax like the callbacks are. Instead they are just listed one after the other, separated by a comma, as if they were regular arguments. See table below for details on the create_parm() function.

How to Use Attribute Codes

There are two arguments in the **create_item()** function that contain attribute codes. In the syntax example above, one is called *attrib* and the other is called *extraAttribute*. Each attribute is actually a combination of a number of individual attribute codes, added together. To determine the value to set in the call to **create_item()**, add the numerical equivalent of all desired codes. An example is shown below with the few codes that might be used in design kit components.

For a more complete list of the commonly used attribute codes, refer to the *"AEL"* manual and use the codes and values from Table 15-1 in the *"AEL"* manual to set the *attrib* argument. Use Table 15-2 in the *"AEL"* manual to set the *extraAttribute* argument. These are stored in separate arguments because the size of the number that can be stored in memory is limited and would be exceeded if the codes from both attributes were stored in one location. Do not mix values from the different tables.

Attribute Code Examples

For the *attrib* argument, the attribute code most applicable to design kit components is the code called ITEM_UNIQUE. This is used for the process include component and it means that only one instance of this component can be placed in a schematic page.

From Table 15-1 in the *“AEL”* manual, the numerical value of that code is 8. Therefore, the **create_item()** function call for a process include component will have the value 8 for the *attrib* argument. For better documented code, you might choose to use the more descriptive code and set the value to ITEM_UNIQUE, from the first column in Table 15-1 in the *“AEL”* manual.

Another code that could be used for process include components is called ITEM_NOT_NETLIST_IF_SUB. This is not listed in the AEL document. It means that if the item is placed in a subcircuit, do not netlist it. In other words, only output it to the netlist if it is placed in the top level of the circuit. This will prevent nested subcircuits which the simulator does not support. See “Using a Subcircuit Model” on page 3-31 for how to avoid nested subcircuits. It also prevent multiple instances from being written to the netlist. ITEM_UNIQUE only ensures that multiple instances are not placed in one schematic page. It does not check other levels of hierarchy.

The decimal value of ITEM_NOT_NETLIST_IF_SUB is 33554432. This can be tested by entering fputs(stderr, ITEM_NOT_NETLIST_IF_SUB); in the ADS command line dialog, which is accessible from the ADS main window. To combine these two codes, you can add the two values and set the attrib argument to 33554440.

For better documented code, you can enter the following:

```
ITEM_UNIQUE | ITEM_NOT_NETLIST_IF_SUB
```

This can also be tested in the command line by entering the following:

```
fputs(stderr, ITEM_UNIQUE | ITEM_NOT_NETLIST_IF_SUB);
```

For information on how to view the output, refer to “Viewing Debug Output” on page 3-7.

Note At this point it is necessary to mention that these descriptive codes are actually hexadecimal codes inside the program (0x010 and 0x02000000, respectively). This is why they must be combined with the '**|**' symbol (**OR**) when the code name is used and not the decimal equivalent. It is beyond the scope of this document to teach hexadecimal arithmetic. Just remember that you must use the **OR** symbol if you are using the descriptive codes. Only add the numbers together if you are using the decimal equivalents.

All other design kit components should not need any of the attributes from table 15-1 in the *“AEL”* manual, so the value 0 or NULL or ITEM_NORMAL should be set. ITEM_NORMAL is not shown in the AEL manual.

For the *extraAttribute* argument, set ITEM_PRIMITIVE_EX to indicate to the simulator that the component is a native component in the simulator. ITEM_CKT_MODEL_EX would be set for a user-compiled model for Analog/RF. ITEM_SP_MODEL_EX indicates a user-compiled model for DSP. Other common codes for the *extraAttribute* argument are given in Table 15-2 in the *“AEL”* manual.

Parameter Definition

This section describes the syntax and arguments for the **create_parm()** function, which is used as an argument in the **create_item()** function, described above. More information is included in Chapter 15 of the *“AEL”* manual.

The syntax for the **create_parm()** command is shown here. The square brackets indicate optional parameters, but are not entered in the actual function call.

```
create_parm(name, desc, attrib, formSet, unitCode, defaultValue
[,cbList]);
```

An example similar to that in the tutorial is repeated here for reference. The callback information is incomplete but is sufficient for demonstration purposes. The parameter descriptions are included in Table 4-2.

```
create_parm("Model", "Model Instance Name", 0, "StdFileFormSet", -1,
            prm("StdForm", "BJTM1"),
            list(callback1, callback2, etc));
```

Table 4-2. The **create_parm()** Parameter Descriptions

Argument	Description
name	Name of the parameter.
desc	Description of the parameter. This text is seen by the user in the edit parameter dialog when placing a component or modifying its parameter values.

Table 4-2. The **create_parm()** Parameter Descriptions

Argument	Description
attrib	Special attribute code. See "How to Use Attribute Codes" above. The list of commonly used parameter attributes is given in the AEL manual in the create_parm() section. Design kit components usually do not need a special attribute code. Set the value to 0. Notes: The code for PARM_NOT_ON_SCREEN_EDITABLE may be used in conjunction with formsets. This will prevent the user from typing in a value that is not in the preset list, but it will also prevent the user from using the up/down arrows to scroll through the values on-screen. If you are defining dependent parameters, as discussed in Parameter Callbacks in Chapter 6, do not use the PARM_NOT_EDITED attribute code. This will prevent the user from editing the parameter, but it will also prevent any AEL callback code from being able to edit the parameter.
formSet	Forms are used as a way to present the user with a list of values. A parameter value is then limited to those defined in a form set. List the name of the form set here. See Forms and Formsets in Chapter 4 of this manual, or Chapter 15 of the AEL manual.

unitCode	The following is the list of unit types for ADS components.
	
	STRING_UNIT = -2
	UNITLESS_UNIT = -1
	FREQUENCY_UNIT = 0
	RESISTANCE_UNIT = 1
	CONDUCTANCE_UNIT = 2
	INDUCTANCE_UNIT = 3
	CAPACITANCE_UNIT = 4
	LENGTH_UNIT = 5
	TIME_UNIT = 6
	ANGLE_UNIT = 7
	POWER_UNIT = 8
	VOLTAGE_UNIT = 9
	CURRENT_UNIT = 10
	DISTANCE_UNIT = 11
	TEMPERATURE_UNIT = 12
	DB_GAIN_UNIT = 13

defaultValue	defaultValue is optional. If specified, it needs to be in the form of a value returned from the prm() function. The prm() function generates an acceptable default value for parameters with different form sets. If a formset is not set for the parameter, use "StdForm" as in the example prm("StdForm", "BJTM1");.
cbList	List of parameter callbacks. See Chapter 6, Parameter Callbacks for details.

Saving the Item Definition

As you learned in the "Adding Demand Loaded Components" on page 3-29, the AEL file that contains the item definitions must follow a certain format. The only restriction is that the **create_item()** call be the last function for each component. The other code that is in the file will be global variables, forms and formsets, and netlist and parameter callbacks. It is recommended that you always present the code in this order. The file *mykit_process.ael*, which is shown in the "Design Kits for RFIC Dynamic Link" on page A-1, is a good example of this format and can be used as a template for development of item definition files. You also learned in the tutorial that item definitions can be combined in one file or be kept separate with one component per file. If you combine the components in one file, make sure that each section looks like the template file and that the **create_item()** function call is always the last piece of code per component in case you choose to create an IDF file for demand-loaded libraries at some point, now or in the future.

Forms and Formsets

The file *mykit_process.ael* in "Example Process Component with Forms and Formsets" on page 4-25 introduces an important topic not covered in the tutorial. It is the use of forms and formsets. This is a method of making user-defined lists that can be presented to the user when they are placing a component on the schematic. The syntax for the function calls to create a formset and a constant form are both very simple. Look at the code sample in "Example Process Component with Forms and Formsets" on page 4-25 and the syntax listed below. More information is available in Chapter 15 of the "*AEL*" manual. Other types of forms are available as well.

create_constant_form(*name*, *desc*, *attribute*, *netlistFormat*, *displayFormat***);**

where:

name is a string representing the form name.

desc is a description of the form.

attribute is an integer, usually 0 for this type of form.

netlistFormat is the format string to netlist the parameter value as. This is blank in the example in the appendix because there is a netlist callback that sets the value. If there were no netlist callback, the string might be similar to the displayFormat string which shows up on schematic.

displayFormat is the format string to display in a schematic.

create_form_set(*name, formName1, formName2,.... formNameN***);**

where:

name is the name of the form set. This is used by the **create_parm()** function.

formName is one or more form names, such as the "name" parameter in the **create_constant_form()** function above.

Note that forms and formsets require the same treatment as all other AEL functions and variables - their names must be unique in the whole ADS environment, including all loaded design kits. So use the same method prescribed for all names in a design kit - prefix them with the design kit, company and/or process name.

Schematic Symbol

When a component is selected from the *Component Palette List* or the *Library Browser*, an instance of the component is attached to the cursor and an outline of the component symbol is displayed as the mouse is dragged across the screen. When the mouse is clicked in the schematic window, an instance of the symbol is drawn on the screen. As a design kit creator, you must create the symbol graphics that are displayed on the schematic.

Typical ADS symbols are one schematic inch wide from the left most pin to the right most pin. The traditional location for pin 1 is on the left. If there is more than one pin on the left side of the symbol, the upper most pin is usually pin 1. Pin 1 is also typically defined as the origin, which is the point where the cursor is when then symbol is dragged across the schematic page.

There are several ways to create a symbol for your design kit.

- "Drawing a New Symbol" on page 4-13 describes one way of creating a symbol using the ADS Symbol Editor.
- "Copying an Existing Symbol" on page 4-15 is another way to create and modify a built-in symbol or a symbol from another design kit or library.
- A third option is to use the IFF export tool from another system such as Cadence DFII and then import the symbol into Advanced Design System. For more information on IFF export and import, refer to the IFF translation documentation for the product you are interested in. The available manuals are
  - "*Translating Cadence Schematics*"
  - "*Translating Mentor Graphics Schematics and Layouts*"

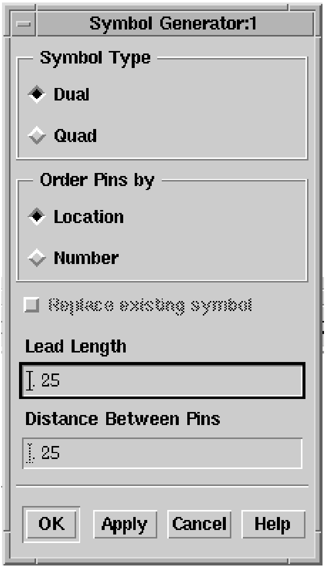
- "*Translating Mentor Graphics Libraries*"

Note It is possible that you will decide to use unmodified built-in ADS symbols in your design kit. In this case, the symbols do not need to be copied to your design kit circuit/symbols directory. You can simply reference the symbol by name in your **create_item()** command. No definition is needed.

Drawing a New Symbol

To draw a new symbol for your design kit:

1. Choose **File > Open Project** to open a project with a new schematic window. For more information on ADS projects, refer to "*Managing Projects and Designs*" in Chapter 2 of the ADS "*User's Guide*".
2. Access the ADS Symbol Editor by choosing **View > Create/Edit Schematic Symbol**. The Symbol Generator dialog box appears.



Symbol Generator:1

Symbol Type: [✓ Dual] Quad

Order Pins by: [✓ Location] Number

☐ Replace existing symbol

Lead Length: 25

Distance Between Pins: 25

3. Close the Symbol Generator dialog and draw the symbol by hand. For more information on creating symbols in ADS, refer to "*Drawing a Custom Symbol*"

in Chapter 10 of the ADS "*User's Guide*" and/or "*ADS Schematic Symbol & Bitmap Creation*" in Chapter 9 of the "*DesignGuide Developer Studio*" documentation.

Note Use of the Symbol Generator is not appropriate for design kit development because the symbols that are automatically generated by the system are just box-like symbols intended for representing a hierarchical design in ADS. For design kits, you will want to create a more meaningful symbol that properly represents the component for which it is a symbol.

Copying an Existing Symbol

To copy an existing symbol for your design kit,

1. Open a schematic window.
2. Choose **File > Open Design** to open the appropriate symbol file. You will find built-in ADS symbols in the directory $HPEESOF_DIR/de/symbols and $HPEESOF_DIR/circuit/symbols. $HPEESOF_DIR is the installation directory for ADS. To copy symbol files from another design kit, browse to the circuit/symbols directory of the design kit. Symbols are stored as .dsn files, also known as design files. Once you have opened the design file, re-save it in the current project directory as the component name.

Detailed steps on this process are provided in the tutorial in Chapter 3, ADS Design Kit Tutorial. For more information on creating symbols in ADS, refer to "*Drawing a Custom Symbol*" in Chapter 10 of the ADS "*User's Guide*" and/or "*ADS Schematic Symbol & Bitmap Creation*" in Chapter 9 of the "*DesignGuide Developer Studio*" documentation.

Component Palette vs. Library Browser

You must decide if your design kit components will be accessed from the component palette, the library browser, or both. You must also decide if your components will be loaded only when used (demand-loaded), to save time at ADS startup, or if the parts will be loaded into memory when the library is loaded.

For a design kit with a limited number of components, selecting a component from the palette is quick because the palette is a permanent part of the schematic window and there are pictures that enable quick identification of the components.

Finding a component on a palette is not efficient if the design kit has a large number of components. In this case, use of the library browser is recommended. The disadvantage is that the library browser resides in a separate window from the schematic page. A distinct advantage is that the browser has search capabilities. When defined with control files, the library browser has other advantages, listed below.

There are two methods for defining components in the library browser. One is a simple AEL function call (**library_group()**) that is usually used in conjunction with the command to create the palette (**de_define_palette_group()**) or when the component is created (**create_item()**). The other method uses library browser control and record files to give more functionality, including the ability to group components in sub-libraries.

Using library browser control and record files is required if components are to be demand-loaded. A benefit of demand-loaded libraries is that you can ship the platform independent binary item definition file (*.idf) with your design kit and avoid shipping the .ael files, thus protecting your code from being modified by the end user. Even if you do not use the .idf file, you can still remove .ael files before shipping your design kit, leaving only the platform-independent compiled versions of those files (.atf files). This is explained more in "Packaging for Distribution" on page 5-2.

The benefits of using the library browser with the control and record files are:

- It allows for a two-tier library presentation (library/sub-library).
- It allows for control over the order in which the components are presented.
- It allows access to additional fields in the library browser.
- It allows for selected components to appear in the library browser when a schematic is open, but not when a layout is open, or vice-versa.
- It is a necessary condition for using demand-loaded components.

The functions mentioned above are defined in the sections below, and the Chapter 3, ADS Design Kit Tutorial provides specific examples of each method.

Component Palette

To define a component palette, use a function call to the ADS function **de_define_palette_group()** in a file called *palette.ael*. This file should be saved in the de/ael directory of your design kit directory structure.

The syntax for this function call as defined in the AEL manual is:

```
de_define_palette_group(winType, dsnType, groupName, groupLabel, position,
item1name, item1label, item1bitmap name[,  item2name, item2label,
item2bitmap name, ...] );
```

The square brackets indicate that the number of items in the palette is variable. The group of three parameters, *item1name*, *item1label*, and *item1bitmap*, are a set and all three have to be defined for each component in the palette. Do not include the bracket in your AEL code. For more information on function syntax and usage, refer to Chapter 15 of the *"AEL"* documentation.

The following values are used for this application:

```
winType = SCHEM_WIN
```

```
dsnType = "analogRF_net" or "sigproc_net"
```

```
groupName = name of palette

groupLabel = descriptive label for palette

position = position is an integer that indicates where the new palette is
added in the list of palettes:

    -1 -alphabetically sort the list of palettes after adding the new
    palette.

    -2 - append the new palette at the end.

    >0 - insert the new palette in the specified position in the list of
    palettes.

item<n>name = name of component as registered in create_item() call

item<n>label = description of component

item<n>bitmap = path to bitmap for the palette button.
```

Important The bitmap name *must not* include the file name extension (.bmp) or it will fail on some platforms. Also note that the second parameter in *dsnType*, used in **de_define_palette_group()**, is a quoted string.

Bitmaps

The bitmap file referenced in the palette definition is a file located in the circuit/bitmaps/pc or circuit/bitmaps/unix subdirectory of the design kit directory structure. The bitmap is used to make the picture icon on the palette, to enable quick selection of a component by visual identification. In addition to a picture, the bitmap can contain a few short words to identify the component and the design kit to which it belongs.

There are different ways to create a bitmap. Windows programs like Paint can be used. The ADS *"DesignGuide Developer Studio"* can also be used and is recommended as it gives access to sample bitmaps and is designed to create bitmaps specifically for use in ADS. The DesignGuide Developer Studio is shipped with ADS and can be used without a license; however, it is not installed by default.

Bitmaps should be created as 32x32 pixels and saved as 16 color bitmaps. If you create your bitmaps on the PC, a unix version needs to be generated from the pc file. The program $HPEESOF_DIR/hptolemy/bin.win32/bmptoxpm.exe can be used for this purpose. The free-ware program *XnView©* can be used as well to save the X windows (unix) version of a bmp file. If you use the *DesignGuide Developer Studio*, you can save both formats directly from the bitmap editor.

Note that the standard way to store bitmaps for an ADS design kit is to give the unix and PC bitmaps identical names and store them in subdirectories called unix and pc, under the circuit/bitmaps directory of the design kit. The .bmp extension can be used for both. The DesignGuide Developer Studio bitmap tool has **SaveAs** commands for both unix and pc. More information is included in Chapter 3, ADS Design Kit Tutorial.

Note The bitmap name as listed in the call to **de_define_palette_group()** *cannot* contain a filename extension such as *.bmp*. The bitmap file name can however include the full path. The presence of a file extension will cause display of the bitmaps to fail on some platforms.

Library Browser

Design kit components can be entered into the library browser in one of two ways. The simplest way is with a function call to **library_group()**. An example of this is given in the tutorial. It is a simple list of the components as defined in the item definition, preceded by two arguments that identify the name and description of the library. The syntax from the AEL manual is:

```
library_group(name, label, item1, item2, ..., itemN);
```

If the component palette has been defined as above in *palette.ael*, this command can be added to that file, or it can be added to the *boot.ael* file defined in "Creating the boot.ael File" on page 3-5.

The other way to define a library for the library browser is by setting up control and record files in the circuit/records subdirectory of your design kit as defined in the next section. This method provides the functionality listed in the section "Component Palette vs. Library Browser" on page 4-15. All library control files (.ctl) in the <*design_kit_name*>/circuit/records directory will be loaded automatically when the library is activated. Record files will be called by the control file when they are required. The control file defines the libraries and the record files list the components in each library.

The Control File

The control file creates the two-tier presentation (library/sub library) in the library browser. The control file uses the Extensible Markup Language (XML) format. The syntax of the control file is shown below. All words and characters shown are required tokens (except the dots). Dots indicate where the information is filled in for each library.

```
<?xml version="1.0" ?>
<LIBRARIES>

  <LIBRARY>

    <NAME>...</NAME>
    <CATEGORY>DL</CATEGORY>
    <SUBLIBRARY>

      <NAME>...</NAME>
      <RECORD_FILES>...</RECORD_FILES>

    </SUBLIBRARY>
    <SUBLIBRARY>

      <NAME>...</NAME>
      <RECORD_FILES>...</RECORD_FILES>

    </SUBLIBRARY>
    <SUBLIBRARY>

      <NAME>...</NAME>
      <RECORD_FILES>...</RECORD_FILES>

    </SUBLIBRARY>

  </LIBRARY>

</LIBRARIES>
```

Sub libraries are optional but if not used, then the RECORD_FILES must be defined for the library. Multiple record files can be specified on one line, separated by a space. The following optional fields can be defined for each library or sub library.

```
  <URL>...</URL>
```

Enter any appropriate web address where the user can find information on the library or component.

```
  <AVAILABILITY>...</AVAILABILITY>
```

Valid settings are *Available*, *Obsolete* and *Not Available*.

The Record File

A record file lists all of the components in a sub library. Each record file must have the extension *.rec*. The format for the record file is similar to that of the control file.

```
<?xml version="1.0" ?>
<COMPONENTS>

   <COMPONENT>

      <NAME>...</NAME>
      <DESCRIPTION>...</DESCRIPTION>
      <VENDOR>...</VENDOR>
      <LIBRARY>...</LIBRARY>
      <PLACEMENT>...</PLACEMENT>
      <AVAILABILITY>...</AVAILABILITY>
      <URL>...</URL>

   </COMPONENT>

</COMPONENTS>
```

Multiple component sections can be defined between the beginning and ending COMPONENTS tags.

Multiple record files can be specified on one line, separated by a space.

The PLACEMENT tag indicates where the component can be used. Valid settings are Both, Layout, Noschematic and Nolayout. This line is optional and the default value is *Both*.

The following optional fields can be defined for each library or sub library.

```
   <URL>...</URL>
```

Enter any appropriate web address where the user can find information on the library or component.

```
   <AVAILABILITY>...</AVAILABILITY>
```

Valid settings are *Available*, *Obsolete* and *Not Available*.

Note The library/sub-library names used in the library control files do not have to match the name of the design kit, as defined in ads.lib. For the library browser, they can be more descriptive.

For more information on setting up library browser files for a design kit, refer to "Adding Components to the Library Browser" on page 3-26.

Demand Loaded Components

As described above, ADS startup time can be reduced if components in a very large library are not loaded until they are needed. This is called *demand-loading*. To accomplish this, all the ael functions related to the component are stored in an Item Definition File (.idf extension). This file is stored in the same directory as the control and records files, the circuit/records subdirectory of the design kit directory.

The .idf file is created by a utility program that is shipped with ADS and is stored in the bin directory under $HPEESOF_DIR, which is the installation directory of ADS. $HPEESOF_DIR/bin must be in your path to use this utility, but it should already be in your path if you are running ADS.

The syntax for running the program is

```
hpedlibgen -list <comp list file> -out <.idf file>
```

where <*comp list file*> is a simple text file listing each AEL file to be compiled. Each AEL file name is listed, one per line.

The format of the AEL files that are included is important. Each component can consist of a **create_item()** function call, as well as parameter or netlist callbacks and other AEL functions. For this process, the **create_item()** call has to be the *last* AEL function call for that component. In other words, the callbacks and all other associated functions must come *before* the **create_item()** function. Since each block of functions will only be loaded into ADS as needed, any common code such as a reusable utility function, which is used by more than one component, must be stored in an AEL file that is not compiled into the .idf file.

"Adding Demand Loaded Components" on page 3-29 includes an example of creating the .idf file.

Model Files

Model files are one way to include simulation data in a design kit. Refer to Chapter 2, Understanding the ADS Design Kit File Structure to determine what form your simulation data will be in. Other methods for including simulation data are discussed in Chapter 6, Additional Parts for ADS Design Kits. The Chapter 3, ADS Design Kit Tutorial includes an example of using a model file, the most common method for RFIC foundry kits.

Model files are typically translated from Hspice or Spectre formats. The ADS SPICE or Spectre Netlist Translator can be used to perform these translations. After translation, the ADS Model Verification Toolkit can be used to set up a suite of verification tests, where the simulation results from both simulators are compared and plotted against each other.

In the design kit structure, model files are stored in the circuit/models directory. These model files contain process variables and model cards or subcircuit models in netlist format which are referenced by the components in the schematic and need to be available to the simulator.

To give the simulator access to these model files, a netlist include component or a custom process component is placed in the schematic. The section on "Adding a Netlist Include Component" on page 3-18 included an example of creating this type of component. When this component is encountered during netlisting, a **#include** statement is written into the netlist.

The **#include** statement is called a pre-processor statement. When the simulator pre-processor reads the **#include** statement, it reads the referenced netlist file and loads the process variables and models which are in the netlist file. Netlist Include and custom process components as well as pre-processor statements are discussed in depth in the following sections.

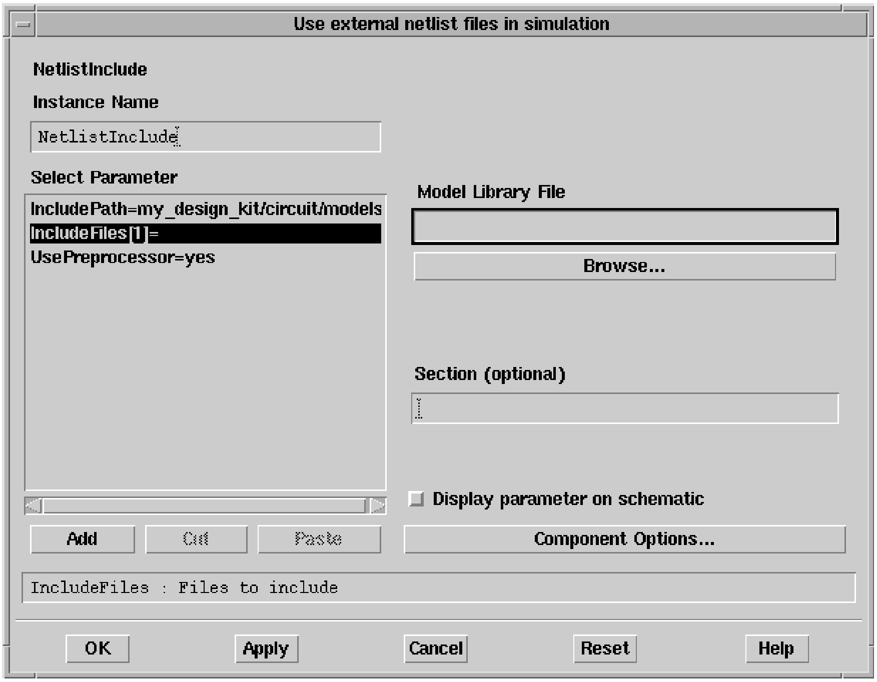
Another pre-processor statement commonly used in model files is the **#define** statement, and the corresponding **#ifdef** and **#endif** statements. These are used to define corner cases for process variations. Corner cases may also be referred to as sections. In a model file, a section is started with a **#ifdef** statement and ended by a matching **#endif** statement. A model file or set of model files will include a number of sections with different process variables and model parameters. To enable a section, a netlist must contain a **#define** statement before the **#include** statement. These pre-processor statements are also covered in the next section.

Netlist Include or Process Component

As mentioned above, the best way to connect the model to the netlisted schematic parts is through the use of a *Process* or *Include* component placed on the schematic. This component does not have any connectivity information but can be used to specify information such as corner cases, as well as the name of the model file to be used for the simulation. A netlist callback on this component will generate a **#include** statement in the output netlist. A netlist callback is an AEL function that is executed during netlisting, the process of traversing a schematic and outputting a file for the simulator. Netlist callbacks are addressed in "Netlist Callbacks" on page 6-14.

Starting in ADS 2002, the *Netlist Include* component is available from the *Data Items* palette. This component is mentioned here to give you an introduction into how the include components work. You can use this while testing your model files, but for your final design kit, it is recommended that you create a custom component to assist your users in attaching the proper file. This was demonstrated in the tutorial in "Adding a Netlist Include Component" on page 3-18.

To use the NetlistInclude component, click the component bitmap in the Data Items palette and place it on the schematic. Alternatively, you can just type *NetlistInclude* in the component history field on the schematic toolbar. Double-click the component in the schematic to display the dialog box. The dialog that is used to enter the parameters for the component is shown in Figure 4-3.



NetlistInclude

IncludePath=my_design_kit/circuit/models
IncludeFiles[1]=
UsePreprocessor=yes

IncludeFiles : Files to include

Figure 4-3. Netlist File Include Dialog

The NetlistInclude component as placed in a schematic is shown in Figure 4-4.

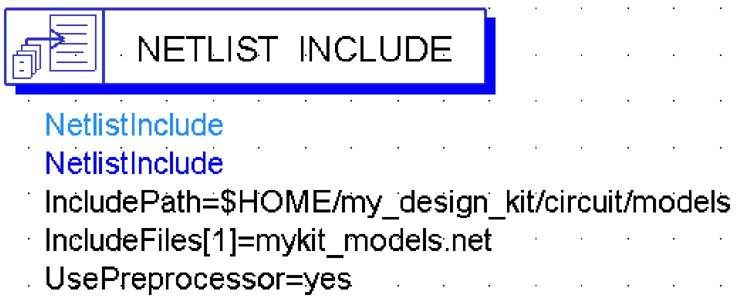

NETLIST INCLUDE
NetlistInclude
NetlistInclude
IncludePath=$HOME/my_design_kit/circuit/models
IncludeFiles[1]=mykit_models.net
UsePreprocessor=yes

Figure 4-4. Netlist Include Component

Only one NetlistInclude component can be placed in a schematic. An error dialog will appear if you attempt to place a second NetlistInclude component.

For this application, only one parameter on the component needs to be filled out. This is the *IncludeFiles* parameter. In the NetlistInclude component dialog, select this parameter with the cursor. On the right hand side of the dialog is a browser button. To demonstrate how this works, use this browser to navigate to the mykit_models.net file that you created in the tutorial. If you did not complete that part of the tutorial, the file can be found in your ADS installation at $HPEESOF_DIR/examples/DesignKit/bjt_dc_prj/design_kit. If you cannot find this path, it is ok for this example to create an empty file called *mykit_models.net*.

When you have browsed to a model file and selected it, note that the *IncludePath* field is automatically filled in for you. You can also manually type in a path or add more paths to the existing one. Each directory can be listed in the same field, separated by space.

Sometimes more than one model file will be included in a design kit. To add more files to be included, select the *IncludeFiles* parameter again and click the **Add** button at the bottom of the Select Parameter list box. A parameter called *IncludeFiles[2]* will be added to the parameter list in the dialog.

The preceding section introduced the concept of a corner case. This is also sometimes called a *section*. A model file will typically be divided up into multiple sections, each representing a process variation that needs to be simulated. It is not possible in the simulator today to automatically simulate all corners in batch mode, so they must be specified individually on the include or process component for each simulation. In the NetlistInclude component edit parameter dialog, type the word *FAST* into the *Section* field on the right hand side of the dialog.

In order to understand the whole picture, let us now create the netlist fragment that will represent the information on this component. There is a short cut to creating the netlist without actually invoking a simulation. Open the Command Line window from the ADS *Main* window by selecting **Options > Command Line**. In the Command field of the command line window type **de_netlist();**. Enter the command using the Enter key or click the **Apply** button to process the command. In a text editor, open the *netlist.log* file in your project directory. It should look similar to the following:

```
#define FAST
#include "$HOME/my_design_kit/circuit/models/mykit_models.net"
#undef FAST
```

The commands that start with the # sign are all pre-processor commands. You can see how the information from the edit parameter dialog gets used in the netlist file. The specified corner case or section is turned on by a **#define** statement and the model file gets referenced by a **#include** statement. It is a good practice to use **#undef** to turn off the section after the file that uses it has been read.

A sample process component which uses forms and formsets to define corner cases by writing **#define** statements to the netlist is include below. Pre-processor commands will be described in more depth in the section following the sample process component.

Example Process Component with Forms and Formsets

The following AEL code can be copied and modified to create your own custom process component. Read the comments after the file for more information on how the file works.

```
/*---------------------------------------------------------------------+/
   FILE         : mykit_process.ael
   COMMENTS     : Component definition :
                    [ global variables ]
                    [ forms and formsets ]
                    [ netlist callback function ]
                    [ parameter callback functions ]
                    item definition
/+---------------------------------------------------------------------*/
/*--- global variables ------------------------------------------------*/
/*--- forms and formsets ----------------------------------------------*/
```

```
/*--- corner cases ----------------------------------------------------*/
create_constant_form("mykit_form_process_best",
                     "Best", 0, "", "Best");
create_constant_form("mykit_form_process_nominal",
                     "Nominal", 0, "", "Nominal");
create_constant_form("mykit_form_process_worst",
                     "Worst", 0, "", "Worst");
create_form_set("mykit_formset_process_corners",
                "mykit_form_process_best",
                "mykit_form_process_nominal",
                "mykit_form_process_worst");
/*--- netlist callback function---------------------------------------*/
defun mykit_process_netlist_cb
(
  cbP,
  cbData,
  instH
)
{
  decl netStrg;
  decl parmH, parmName, parmFormName;
  /*--- #ifdef --------------------------------------------------------*/
  netStrg = "#ifndef MYKIT_PROCESS\n#define MYKIT_PROCESS\n";
  /*--- corner case/resistance -----------------------------------------*/
  parmH = db_first_parm(instH);
  // this while loop isn’t necessary since there is only one parameter
  // but it is shown here as an example for the user
  while (parmH != NULL)
  {
    parmName = db_get_parm_attribute(parmH, PARM_NAME);
    if (parmName == "CornerCase")
{
      netStrg = strcat(netStrg, "; corners\n");
      parmFormName = db_get_parm_attribute(parmH, PARM_FORM_NAME);
      if (parmFormName == "mykit_form_process_best")
      {
        netStrg = strcat(netStrg, "#define MYKIT_BEST_SECTION\n");
      }
      else if (parmFormName == "mykit_form_process_worst")
      {
        netStrg = strcat(netStrg, "#define MYKIT_WORST_SECTION\n");
      }
      else
      {
        netStrg = strcat(netStrg, "#define MYKIT_NOMINAL_SECTION\n");
      }
    }
    parmH = db_next_parm(parmH);
  }
/*--- device models ---------------------------------------------------*/
  netStrg = strcat(netStrg, "; models\n");
  netStrg = strcat(netStrg, sprintf("#include 
'%s/circuit/models/mykit_models.n
et'\n",
                              MYKIT_PATH));
  /*--- #ifdef --------------------------------------------------------*/
  netStrg = strcat(netStrg, "#endif\n");
  /*--- return to calling function ------------------------------------*/
  return(netStrg);
}
/*--- item definition ----------------------------------------------------*/
create_item(
  "MYKIT_PROCESS",                       // name
  "Process Include",                     // description label
  "MYKIT_PROCESS",                       // prefix
  "Process Include",                     // description label
  "MYKIT_PROCESS",                       // prefix
  ITEM_UNIQUE,                           // attributes
  -1,                                    // priority
  "MYKIT_PROCESS",                       // iconName
  standard_dialog,                       // dialogName
```

```
    NULL,                                 // dialogData
    ComponentNetlistFmt,                  // netlist format string
    NULL,                                 // netlist data
    ComponentAnnotFmt,                    // display format string
    "SYM_MYKIT_PROCESS",                  // symbol name
    no_artwork,                           // artwork type
    NULL,                                 // artwork data
    ITEM_PRIMITIVE_EX                     // extra attributes
    ,list(dm_create_cb(ITEM_NETLIST_CB, "mykit_process_netlist_cb",
                       "", TRUE))         // netlist callback
    ,create_parm(                         // parameter
      "CornerCase",                       // name
      "Corner case selection",            // label
      PARM_DISCRETE_VALUE,                // attrib
      "mykit_formset_process_corners",    // formSet
      UNITLESS_UNIT,                      // unit code
  prm("mykit_form_process_nominal")       // default value
      )
    );
```

The process component shown here only has one parameter, for corner case selection. The three corners are *Best*, *Nominal* and *Worst*. A *form set* and three *constant forms* are used to define the list of possible values that are presented to the user when the component is placed in the schematic window. The selected case is used to output the proper **#define** statement in the netlist. This corresponds to a section of the model file, which must be offset by a **#ifdef**/**#endif** block.

Table 4-3 is the skeleton of the models file and the netlist file that are produced by the netlist callback **mykit_process_netlist_cb()**. Inside of the section of the model file will be process parameters and the model with the parameters corresponding to the nominal case. Similar sections will be defined in the model file for the best and worst cases.

Table 4-3. Model File Sample

```
model file mykit_models.net

#ifdef MYKIT_NOMINAL_SECTION
.
.
.
#endif

netlist file from schematic
.
.
.
#define MYKIT_NOMINAL_SECTION

#include "my_design_kit/circuit/models/mykit_models.net"
.
.
.
```

Note that the model file in this example is hard coded as *mykit_models.net*. This would be changed to represent the model file or files in your design kit. The global variable MYKIT_PATH is a path variable that will have to be defined with AEL code.

The #include Pre-processor Command

To understand how pre-processor commands work, it is helpful to understand the whole simulation process. When you click the **Simulate** menu pick, three things happen.

- First, the schematic is traversed in a process called *netlisting* and a netlist file is written in the project directory. This file is saved with the name *netlist.log*.
- Next the simulator pre-processor reads the netlist file. If any pre-processor commands such as **#include** or **#define** are found, they are processed before the netlist is passed to the simulator. So the simulator never actually sees the **#include** statement. Instead, the contents of the file are read and passed to the simulator. It is especially important to understand this with respect to the **#define** statement and for the discussion of limitations of the pre-processor commands.
- The final step is the actual simulation of the information that is now loaded into the simulator.

The **#include** statement is fairly easy to understand. A path and filename are supplied and when that statement is read, the file is read and passed to the simulator. The following list of limitations should be read and understood before you continue.

- If the design kit user is attempting to do remote simulation or parallel simulation, the included files need to be present on the remote machine in the same location that they are on the local machine. This can be accomplished by NFS mounting the disks, which will avoid potential problems caused by copying files around, such as losing edits made in the remote location. If NFS mounting is not possible, a netlist callback could be written that would read the referenced file and output it directly into the netlist, instead of writing the **#include** line into the netlist.
- If an included file has subcircuit models or subcircuits of any type (which start with the command *define*), the process or include component that generates the **#include** statement must be placed at the top level of the hierarchy in the design. Nested defines are not allowed, and placing this type of component into a lower level of hierarchy will result in a define statement within a subcircuit definition. If an included file contains model cards and variables only, the file can be included from any level of hierarchy.
- The name of an included file, as listed on a process or include component, cannot be a variable reference.
- No tuning, sweeping, optimization or yield can be performed on the data in the included file. Additionally, the information in an included file is not available for back annotation on a schematic or other similar functions. This is an acceptable limitation since this information can be thought of as read-only.
- The order of included files cannot be controlled if multiple separate include components are placed. The component called *NetlistInclude* is designed to enforce this restriction. Only one NetlistInclude component can be placed, but it takes a list of files and retains the order of the files when outputting a **#include** line to the netlist.

The #ifdef and #define Pre-processor Commands

As introduced above, **#ifdef** is used to define a corner case or section in a model file. A section can contain process variables, model cards, subcircuit definitions or a combination of these, to represent a process variation. Typical names for corner cases are FAST, NOMINAL and SLOW. A model file containing these sections would look like this:

```
#ifdef NOMINAL
```

[process variables and model parameters for the NOMINAL case would be included here]

```
#endif
#ifdef FAST
```

[the same process variables and model parameters would be included here, but with different values to simulate the FAST process variations.]

```
#endif
#ifdef SLOW
```

[the same variables and models would be included here again, but with values to simulate the SLOW process variations.]

```
#endif
```

Remember from above that the netlisted schematic that would include this model file contains the lines shown below. Note that the sample *mykit_models.net* file does not actually contain these sections at this time.

```
#define FAST
#include "C:/my_design_kit/circuit/models/mykit_models.net"
#undef FAST
```

The order in which all this information is processed is as follows:

1. Netlist file *netlist.log* is created from the schematic. The **#define** and **#include** statements are generated from the include component.
2. The simulator pre-processor reads the *netlist.log* file. FAST is defined and the include file *mykit_models.net* is read.
3. When the model file is read, the NOMINAL and SLOW sections are completely disregarded. Only the FAST section is read.

It was mentioned above that corner cases cannot be swept automatically in the simulator. This is because, based on what was shown in step #3 above, the simulator only has knowledge of one corner section at a time. The others are filtered out by the pre-processor. The user has to turn on each section individually and save the datasets for comparison.

The last concept that needs to be explained is how to use a custom process include component to make it easier for the design kit user to perform these simulations. First of all, the netlist file name can be hard-coded into a custom component. In other words, the user will not have to enter any file name. The path is known because the model file exists in the circuit/models directory of the design kit and the file name is written into the AEL code that generates the netlist. This AEL code is called a netlist callback and is explained in "Netlist Callbacks" on page 6-14.

In order to make it easier for the customer to select which corner case to simulate, the section names are presented to the user in a list, of which one at a time can be selected. To generate this list, a form set is defined in AEL code and then referenced in the item definition, also AEL code. Forms and formsets are defined in "Forms and Formsets" on page 4-11. Item definitions are covered in "Item Definition" on page 4-4. The best way to complete your understanding of these concepts is to return to the Chapter 3, ADS Design Kit Tutorial, where two examples of custom process include components are given.

Any custom process or include component, that is required by other components in a design kit, should be listed at the top of the palette, since the user must place it in the schematic when using any other components in the design kit. In fact, it is so essential that this process component be included in a schematic, that some design kit developers will add a netlist callback to any device component that refers to a model in the netlist model file. This callback can traverse the schematic and warn the user if a process component has not been placed, or it can simply place the process component automatically.

Model Naming Limitations

As mentioned earlier, since ADS has one global name space, all components and variables in a model file must be unique, to avoid colliding with another with the same name if another file is included at the same time. This can happen, for example, when designing a multi-chip module if components from more than one design kit are used in the same design.

This requirement applies to model names, subcircuit names and global variable names. Components or variables (parameters) inside of a subcircuit are considered *local* and are protected automatically. *<foundry>_<process>_<name>* is the recommended naming convention for all global items in an RFIC foundry design kit. At the present time, the netlist translators do not facilitate this process, so it must be done manually or with a script run on the model file after the translation is complete.

The ads.lib Template

The *ads.lib* file is a design kit control file that tells Advanced Design System which design kits to load. A template of this file must be included in the design_kit subdirectory of the design kit directory structure. When a design kit is installed, a copy of the records line of the template file is made and stored outside of the design kit. The location of the file controls who has access to the design kit when ADS is started, as described below.

The format for the records line of the ads.lib template file is a single line with 4 fields, separated by vertical bars.

```
kitname | path to design kit | path to kit boot file | kit version
```

Table 4-4 provides a description for each of the fields shown.

Table 4-4. The ads.lib Fields

kitname	The ‘name of the design kit’. This is also a global variable available in any custom design kit AEL code. The name of the variable is identical to the name if the design kit as registered in the ads.lib file.
path to design kit	In the template file, the second field is left blank or as shown in this template. When a design kit is installed, the template is read, the path is determined and a copy of the line entered in an ads.lib file somewhere else on the system, where it can be referenced during future ADS sessions at startup time to instruct the system which design kits to load. The path is inserted in the copy but not back into the template.
path to boot file	This is an optional value. If specified, it is the relative path from the top of the design kit structure to an AEL file which has special instructions for loading the design kit, and potentially other AEL files. More details on the boot file are given in the next section of this chapter.
kit version	This is the official location for recording the version of a design kit. It is a string. Make sure you update it if you release a new version of the design kit. Any changes to a design kit should trigger a new version number to maintain the integrity of the complete set of files.

To insert a comment into the template ads.lib file, begin the line with a pound sign (#). Comments will not be copied with the records line.

Ultimately, *ads.lib* files can exist in the any of the locations listed in Table 4-5. Note that these locations are referred to as levels in the ADS Design Kit user interface.

Table 4-5. The ads.lib File Locations

Level	Directory	Description
SITE LEVEL	$HPEESOF_DIR/custom/design_kit	An ads.lib file in this location on a networked system lists all the design kits that are available for all users.
USER LEVEL	$HOME/hpeesof/design_kit	An ads.lib file in this location lists design kits that only the user has access to.
STARTUP LEVEL	Startup directory	An ads.lib file in this location lists design kits that will only be available if ADS is started in this directory.
PROJECT LEVEL	Project directory	An ads.lib file in this location is read when the project is opened. Any design kits listed in it will be made available for designs in the project.

With this capability, a user in a networked environment can have access to a specific design kit without exposing all engineers to it or without requiring the assistance of an administrator with root permissions to install it. Another benefit is that a site librarian can manage a set of libraries that are accessible to all engineers without any effort from the engineers. These are some typical ways that a system is configured.

Another way to use this multi-layered functionality is for an end user to have a set of ads.lib files in different directories that refer to different processes. By starting ADS in the proper directory, the related design kits will be loaded for that session. An advanced CAD manager may even choose to set up a system of scripts that controls the *ads.lib* files for the end user to further tailor the environment on the fly for a specific manufacturing process.

Note To start ADS from a specific directory on unix, **cd** to that directory. On a PC, multiple shortcuts can be created and the shortcut property “**Start in**” can be set to different startup directories.

Even if a design kit is listed in the system *ads.lib* file for access by all engineers, the engineer can still disable that configuration by telling ADS to only load *ads.lib* files in the local area. This can be set up by choosing the **DesignKit > Setup Design Kits** menu pick from the ADS *Main* window. The current design kit configuration can be viewed using the *List ADS Design Kits* dialog by choosing **DesignKit > List Design Kits** menu. For more information, refer to "Viewing Configuration Files and Variables" on page 8-8.

If a design kit is listed in more than one *ads.lib* file, those specified in the $HOME location will have precedence over those under $HPEESOF_DIR, and those specified in the startup directory will take precedence over those in the $HOME location. Additionally, each project directory may have its own *ads.lib* file. This file will be read when a project is opened, and any design kits specified in that file will be loaded. If a design kit by the same name is already open, it will be overwritten with the new information.

Closing a project to open a different project will not unload design kits opened in that project since design kits cannot currently be unloaded without restarting ADS.

Even though all design kits are intended to be able to coexist in ADS, there may be cases where this is not possible. Some older design kits may not comply with the standard format. Additionally, ADS has a requirement that all components have unique names in ADS, so names cannot be reused between design kits. If these types of problems are encountered, they can usually be overcome by enabling and disabling design kits as needed through the user interface or by manually editing ads.lib files and the design_kit.cfg file. Additionally, some custom AEL in older, non-standard design kits may have problems co-existing with the current built-in design kit software. Chapter 8, Setting Up Design Kit Software and Menus includes instructions for temporarily disabling the new software for this rare case.

AEL Code for Loading a Design Kit

As mentioned in the previous section, a design kit will contain a boot file that helps the system load the design kit. The name of this file is recorded in the *ads.lib* file so there isn't a reserved name for the file that is searched. However, the name *boot.ael* is typically used in most ADS design kits today and the file resides in the de/ael subdirectory of the design kit directory structure.

A boot file can be used to load the AEL item definitions for all components in a design kit, as well as to set up the palettes. The Chapter 3, ADS Design Kit Tutorial gives an example of this use. Additionally, the boot file may be used to define some global variables or functions specific to the design kit. These are usually for advanced functionality and specific details are not included in this document.

Including a boot file is optional. If a boot file is not specified in the *ads.lib* file, component AEL will still be loaded and the library browser will still have knowledge of the library and sub library definitions, as described earlier in this chapter. The program will first look for an item definition file (.idf) in the circuit/records directory. If this is found, all the item definition AEL functions will be loaded from the .idf file. If no .idf file is present, any application extension language (.ael) or compiled AEL files (.atf) stored in circuit/ael will be read. Either way, the control (.ctl) and records (.rec) files from the circuit/records directory will be read.

As mentioned in the previous section, the design kit name as stored in the first field of the ads.lib template becomes a global AEL variable containing the path to the design kit directory. This can be used in AEL code to find other files in the design kit. See the boot.ael and palette.ael sections of the tutorial for a specific example of how this variable is used.

The about.txt File

A file named *about.txt* should be supplied and stored in the doc directory of the design kit structure. This is required for version tracking and will assist Agilent Technologies customer support. The file will contain information such as the design kit version and date created. Process and component information can be included, as well as source information if the models were translated from another library. Revision history, support contact information or any other details can be listed if desired.

A template for this file is shown in Table 4-6. A future version of the design kit software may have a menu pick to give the user easy access to this information.

Table 4-6. Design Kit about.txt Template

Name:	
Version:	
Date:	
Description:	
Revision History:	

For more information on design kit versions, refer to "Assigning a Version" on page 5-1.

The Example Project

The quickest way to help your customers start using your design kit is to provide a sample design that they can work from. This is a simple way for them to see that a design kit is installed properly and verify that they can run a simulation. It is also a way for you to show some special features of the design kit.

To add the example project to the design kit, first create and save a small test circuit, including a process or include component if required. Then simulate the circuit and save the data display. More test circuits can also be saved if desired. When all test designs have been saved, archive the project using the **File > Archive Project** menu pick in the ADS *Main* window. Copy the archived project to the examples directory in the design kit file structure. The tutorial steps in Chapter 3, ADS Design Kit Tutorial can also be referred to for help in building an example project and storing it with the sample design kit.

Chapter 5: Completing the Design Kit

This chapter describes the process of finalizing an ADS Design Kit. Verification is an essential step in design kit creation if model files were translated from another simulator format. There is a standard method for packaging and distributing design kits that should be adhered to, and understanding the requirements of design kit support is also important. This chapter is mostly aimed at design kits that are being developed for distribution outside of the organization where they were developed.

Verifying a Design Kit

If a design kit contains translated models, verification of those models is an essential part of ADS design kit development and potentially the most time consuming part of creating a design kit. This is because there can be significant differences between how simulators work, due to the use of different equations or unique extensions to a base technology.

An engineer working on verification must understand the details of the model very well, and must take the time to understand how to correct for simulation differences. To assist in this verification process, Agilent Technologies has developed a model verification tool. Results of the verification can be saved and presented to your customers to give them confidence in the models and the design kit.

Assigning a Version

Each release of a design kit must have a version assigned to it for the design kit to qualify as following the standard. The version can be any string of your choice, but it must be registered in the template *ads.lib* file. For more information on the ads.lib file, refer to "The ads.lib Template" on page 4-33.

This version must be changed any time even one file in a design kit is changed, and the whole design kit should be repackaged and shipped as a single unit, to maintain the integrity of the complete set of files. This is very important for tracking down customer problems. If a customer has a problem with a design in ADS, the Agilent EEsof-EDA Customer Support department cannot investigate the issue without installing the exact version of the design kit that the customer has installed.

It is highly recommended that a revision control software package is used to store the design kit files during development and after release. This type of software tags each individual file with a version. The complete set of files can then be tagged at a release with a tag that is related to the version of the design kit seen by users.

Packaging for Distribution

To distribute your design kit, create a zip archive from the files. Zip is the only format recognized by the ADS design kit installation software. Zip and unzip are shipped as a standard part of ADS. The files are in the $HPEESOF_DIR/bin directory, including documentation in the zip.doc file. If your design kit is packaged in anything other than a .zip file, you must include installation instructions with the design kit and be prepared to support your customers if they have installation problems.

The design kit standard does not extend into the distribution method. It is up to each company to decide the best method for distributing its design kits. The typical method for foundry design kit distribution is from the foundry web page.

Once you have created and tested your design kit, there are a couple more steps you can take prior to zipping up the contents of the design kit.

1. Remove all .ael files. Check the de/ael and circuit/ael directories. If there is a .atf file for every .ael file, you can remove the .ael files. This will prevent users from modifying your code and changing the behavior of your design kit. It also hides the code you have built into your callbacks. One exception to this is the *palette.ael* file. Some users prefer to control the ordering of their palettes themselves. Since there is no user interface to control the palette configuration, setting the palette location in the palette.ael file is the only way to do this. There is no harm in shipping the palette.ael file.

2. Remove the circuit/ael directory. Only do this if you are using demand-loaded components and have combined all the ael files in this directory into a single .idf file.

3. Make sure that none of the files in your design kit are read-only. All files must have full write permission or the unzip procedure may fail. To set the write permission on all files:

   On unix, from the directory at the top of the design kit, enter the command:
   ```
   chmod -R 777 *
   ```

   On PC, open the *Windows Explorer* file browser and click the right mouse button on each directory and filename. Select *Properties* and ensure the *Read-only* attribute is unchecked.

4. Zip the design kit, including the top level directory. Make sure before you start the zip that you are pointing to the top level directory of the design kit, the one which bears the name of the design kit. If you are performing this process from the command line in a DOS or unix shell, cd to the directory above the design kit. To create the zip file *my_design_kit.zip*, enter the following command in a DOS shell or unix window:

   ```
   zip -r my_design_kit my_design_kit
   ```

5. To test the result, copy the zip file to a different directory and unzip it. The unzipped image should contain only one directory at the top level. All subdirectories will be one level below that (see Figure 5-1).

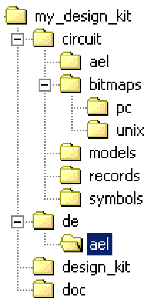


Figure 5-1. Design Kit Directories

Supporting a Design Kit

The creator and supplier of a design kit is responsible for informing the users of the kit how to get help with the kit, including installation, component and model or data file problems. Agilent Technologies customer support can help with general setup issues or simulation issues, but if the problem is specific to the design kit, the support personnel will need access to the design kit.

Your company is expected to have a dedicated resource for this purpose. This person will handle the support calls first. If the problem is determined to be with the ADS environment or simulator, then the call and the design kit can be forwarded to the Agilent EEsof EDA customer support department. The customer should not contact Agilent Technologies directly, since they will not have permission to share the design kit with the Agilent EEsof EDA customer support department. Additionally, if your customer works directly with Agilent Technologies and it is discovered that the problem is in the design kit, then you may not get the feedback that you need to make the corrections to the kit.

Chapter 6: Additional Parts for ADS Design Kits

This chapter describes additional parts that may be added to a design kit. The basic parts were described in Chapter 4, Basic Parts of an ADS Design Kit. The parts described in this chapter are used to provide extra functionality to a kit.

This chapter is divided into two sections. The topics in the first section are covered sufficiently that you should be able to implement the functionality in your design kit. The topics in the second section include a description of capabilities that the system has to offer, but it is beyond the scope of this document to cover them in depth at this time. Contact Agilent EEsof-EDA's Solution Services organization for assistance in implementing these features in your design kit.

Adding Simulation Data to a Design Kit

In Chapter 4, Basic Parts of an ADS Design Kit, you learned about supplying simulation data in the form of an included netlist file which contained model and parameter information. This is the typical style for an RFIC design kit. Other methods of supplying simulation data are used by design kits which serve different technologies. These methods are described in the following section.

S-Parameter and MDIF data

ADS schematics can include components such as *SnP*, *S2P*, *MDIF* and *DataAccessComponent*. These components point to external data files. The file browser on these components uses the DATA_FILES configuration variable to find all data files in that path. If you use the browser to locate the files, be sure to manually strip off the full path to the selected file. You do not want your customers to receive files with a hard-coded path to files on the machine where the design kit was developed.

When a design kit is loaded, the circuit/data subdirectory under the design kit directory is added to the DATA_FILES path variable automatically. The simulator will then be able to find the data file in the path. Make sure the names of your data files are unique to ensure that the simulator finds only one file with the given name. Just as has been done with all other names in your design kit, it is good practice to prefix the file name with the name of the company and/or process.

Root Model Files

The ADS component HP_FET points to a model called HP_FET_Model. This model references an external data file, which is a text file generated from IC-CAP. If your design kit uses Root model files, the same process is followed as described above for S-parameter and MDIF data files. The file should be stored in the circuit/data subdirectory of the design kit, and the DATA_FILES path will be automatically extended so that the simulator will find the file. The file reference on the component should be the file name only with no path information included, and the file name should include some reference to the company or process, to ensure it is unique between all design kits.

User Compiled Models

Starting in ADS2002, user compiled models can be distributed as dynamically linked libraries (.dll file on PC) or shared libraries (.sl or .so files on unix platforms). This is especially important for design kit developers and users because it enables the design kit user to access custom models from multiple design kits simultaneously. To create a user compiled model for a design kit, use the schematic menu pick **Tools >User-Compiled Model** and the associated documentation "*Analog/RF User-Defined Models*". Be sure to make your component name specific to your design kit so it remains unique when used with other design kits.

When the model has been compiled into a dynamically linked library (with file extension .dll, .sl or .so), copy it and the associated index file *deviceidx.db* file from the networks directory of the current project to the bin/$ARCH subdirectory of your design kit. To determine the proper subdirectory of bin, run the program *hpeesofarch*, which resides in $HPEESOF_DIR/bin. For ADS2002, the expected values on the different platforms are win32, hpux10, sun57 and aix4. If you add more files to the directory, you must regenerate the index file by running *hpeesofsim -X* in that directory. This does not require a license but the appropriate shared library/DLL environment variables must be set. For more information on setting the shared library/DLL environment variables, refer to Appendix E of the "*RFIC Dynamic Link Library Guide*".

When a user enables a design kit with a dynamically linked user compiled model, the proper design kit directory is added to the path variable EESOF_MODEL_PATH, which is read by the simulator from the simulator configuration file hpeesofsim.cfg to locate required custom models.

To complete your design kit, copy the project to each supported platform, regenerate the dynamic library, and copy the files to the design kit.

Parameter Callbacks

A parameter callback is an AEL function that is executed automatically when a component parameter is modified in the schematic editor. The function is provided by the design kit developer but must contain the specified arguments and must return the exact value type that is documented. The purpose of parameter callbacks is to provide a means of controlling the value of component parameters that depend on the values of other parameters of the same component. It is expected that you have at least a minimal understanding of programming terms to comprehend this section.

Adding a Callback to a Parameter Definition

Parameter callback functions are associated with individual parameters of a component by means of information in the call to **create_parm()** for the specific component's parameter. This is done by the addition of a optional callback list.

For example, a **create_parm()** call without a parameter callback function might read as:

```
create_parm("X",                     // parameter
            "Unknown value",         // label
            68608,                   // attribute
            "StdFormSet",            // formset
            0,                       // unit code
            prm("StdForm", "10.0")), // default value
```

Examples of the **create_parm()** call without a parameter callback function are also given in Chapter 3, ADS Design Kit Tutorial.

The same parameter information with an associated parameter callback function might read:

```
create_parm("X",                     // parameter
            "Unknown value",         // label
            68608,                   // attribute
            "StdFormSet",            // formset
            0,                       // unit code
            prm("StdForm", "10.0"),  // default value
            list(dm_create_cb(       // callback list
                  PARM_MODIFIED_CB,  // callback type
                  "cb_funct_name",   // callback function name
                  "",                // clientData
                  TRUE))),           // callback enable
```

Parameter callback function information is incorporated into the call to **create_parm()** as a **list()** of calls to the function named **dm_create_cb()**. In the example here, there is only one such call to **dm_create_cb()**, but there could be multiple calls as separate entries of the list. This way, a parameter could be associated with a number of independent callback functions.

The function **dm_create_cb()** takes four (4) arguments:

- For parameter callback functions, the first argument must be PARM_MODIFIED_CB. This is the callback type.
- The second argument – a quoted string – is the name of the AEL function to be invoked when the associated parameter is modified in the schematic editor.
- The third argument – called clientData – is passed to the callback function. An example of a use of this argument is included in "Writing the Parameter Callback Function" on page 6-5. This argument can be an empty string as shown above if there is no extra information to be passed to the callback function. The parameter data is passed automatically.
- The fourth argument - TRUE or FALSE - is used to enable or disable the callback function association.

For more information on the **dm_create_cb()** function, refer to chapter 15 of the "*AEL*" documentation. The examples on the following pages will also help you understand parameter callbacks.

Writing the Parameter Callback Function

The section above described how to add a parameter callback when creating a parameter. Now the actual callback function needs to be written. This is the AEL code that is automatically executed when the component parameter is modified in the schematic editor.

The function declaration has a predefined set of three arguments, *cbP*, *clientData* and *callData*. The *callData* is used to access all parameters on the component so you can use the values of one or more independent parameters to calculate the value of a dependent parameter. *clientData* is a string that was set when the callback was added to the parameter. The *clientData* string gives you the flexibility to define callback functions in a couple different ways.

The first method used to define callback functions is to have a separate function for each parameter on a component. In this case, you might not need to set *clientData* at all. When it is being executed, the AEL function will always know which parameter was being modified when the callback was triggered.

The second method for defining callback functions is to have only one function for each component. The advantage is that it combines all the code related to the component in one location. It also allows reuse of code, which makes it easier to maintain the code, since changing it in one place changes it for all cases. To use this method, you will need to supply an identifier so the function knows which parameter is being modified. This identifier is the string passed in as *clientData*.

The basic structure of a parameter callback function is shown below, with examples of both of these methods on the following pages.

```
defun cb_funct_name(cbP, clientData, callData)
{
  decl dependentParmData = NULL;
  //
  // additional declarations
  //
  //   and
  //
  // callback function code
  //
  return dependentParmData;
}
```

This structure should be used as a template, substituting an appropriate name for *cb_funct_name* and replacing the section marked as comments with parameter declarations and AEL code. Note that it is important to maintain the illustrated declaration, initialization, and return of the variable *dependentParmData*. Examples on the following pages will clarify this.

Note It is OK to change the argument names or *dependentParmData* variable name, as long as it is done consistently throughout the function.

The arguments of the function are:

- *cbP* - This is a pointer to the function itself. There will not be any reason to use this argument.
- *clientData* - This can be any information in string format. It is set when the callback is declared in the **dm_create_cb()** function.
- *callData* - This is the parameter information for the component being edited. Examples of how to access and modify the parameter values are given in the following pages.

It is not necessary to understand the structure of this information, since access functions are provided to extract information about specific parameters.

Callback Example - One Function per Parameter

As an example of a callback that is filled out and functional, consider a component (perhaps a subcircuit model) that has three parameters A, B, and C. Further, assume that the following relationships are to be established among these three parameters:

- When the value of either parameter A or B is modified, the value associated with parameter C is to be adjusted so that it is equal to the sum of A and B;
- When the value of parameter C is modified, the values associated with parameters A and B are to be adjusted so that they are each equal to C/2.

One way of programming these relationships is shown in the following AEL code. First is a code fragment shows part of the item definition AEL for the parameters of the component:

```
create_parm("A",                      // parameter
            "A parameter value",      // label
            68608,                    // attribute
            "StdFormSet",             // formset
            0,                        // unit code
            prm("StdForm", "10.0"),  // default value
            list(dm_create_cb(        // callback list
                  PARM_MODIFIED_CB,  // callback type
                  "a_modified_cb",   // callback function name
                  "",                // clientData
                  TRUE))),           // callback enable
create_parm("B",                      // parameter
            "B parameter value",      // label
            68608,                    // attribute
            "StdFormSet",             // formset
            0,                        // unit code
            prm("StdForm", "10.0"),  // default value
            list(dm_create_cb(        // callback list
                  PARM_MODIFIED_CB,  // callback type
                  "b_modified_cb",   // callback function name
                  "",                // clientData
                  TRUE))),           // callback enable
create_parm("C",                      // parameter
            "C parameter value",      // label
            68608,                    // attribute
            "StdFormSet",             // formset
            0,                        // unit code
            prm("StdForm", "10.0"),  // default value
            list(dm_create_cb(        // callback list
                  PARM_MODIFIED_CB,  // callback type
                  "c_modified_cb",   // callback function name
                  "",                // clientData
                  TRUE))),           // callback enable
```

Next, suitable callback functions are:

```
defun a_modified_cb(cbP, clientData, callData)
{
  decl dependentParmData = NULL;
  decl a_mks = pcb_get_mks(callData, "A");
  decl b_mks = pcb_get_mks(callData, "B");

  dependentParmData = pcb_set_mks(dependentParmData,
                                                "C", a_mks + b_mks);
  return dependentParmData;
}
defun b_modified_cb(cbP, clientData, callData)
{
  decl dependentParmData = NULL;
  decl a_mks = pcb_get_mks(callData, "A");
  decl b_mks = pcb_get_mks(callData, "B");

  dependentParmData = pcb_set_mks(dependentParmData,
                                                "C", a_mks + b_mks);
  return dependentParmData;
}
defun c_modified_cb(cbP, clientData, callData)
{
  decl dependentParmData = NULL;
  decl c_mks = pcb_get_mks(callData, "C");

  dependentParmData = pcb_set_mks(dependentParmData, "A", c_mks/2.0);
  dependentParmData = pcb_set_mks(dependentParmData, "B", c_mks/2.0);
  return dependentParmData;
}
```

The three functions above are very similar. The third one, **c_modified_cb()**, will be used to provide a line by line description of the details of code which retrieves and sets parameter values. It is expected that the reader already understands basic programming concepts.

line 1: `decl dependentParmData=NULL;`

dependentParmData is a variable that will be used to collect the new parameter values. It must be set to NULL to start with so no garbage gets attached to the component.

line 2: `decl c_mks = pcb_get_mks(callData, "C");`

As described in the previous section, *callData* contains the starting values of all the parameters on the component. **pcb_get_mks()** is used to get the numerical value of any parameter on the component. In this case, it is the value of *C*, which the user just set in the schematic, which is being retrieved. It was the modification of this value in the schematic that caused this code to be executed, or *triggered the callback*.

The syntax for this function is:

```
mksValue = pcb_get_mks(callData, paramName);
```

Where:

callData is the third argument of the callback function, if the callback is of type PARM_MODIFIED_CB.

paramName is the name of the parameter to get the value from. This must be a quoted string.

mksValue is the requested value, returned in MKS (unscaled) units.

line 4: `dependentParmData = pcb_set_mks(dependentParmData, "A", c_mks/2.0);`

Note that *dependentParmData* is both passed into this function and returned from it. It needs to be reset because it is accumulating the values for the dependent parameters. A and B are dependent on the value of C in this example.

The syntax for this function is:

```
paramData = pcb_set_mks(paramData, paramName, value)
```

Where:

paramData is a structure containing parameter data. It is NULL the first time it is called. In addition to this variable being a parameter to this function, the value returned by this function must also be assigned to it.

paramName is the name of the parameter to set the value of. This must be a quoted string.

value is the new value in MKS (unscaled) units.

line 5: `return dependentParmData;`

A callback of type PARM_MODIFIED_CB, which these are, must return the collected parameter information, which is stored in dependentParmData.

The function pair **pcb_get_mks()** and **pcb_set_mks()** is aware of scale factors that have been associated with a parameter's value. That is, if a resistance is specified as R=1kOhm, then **pcb_get_mks()** returns a value of 1000. Similarly, a value of 2000 supplied as the third argument of **pcb_set_mks()** for the same parameter results in R=2kOhm.

Not all parameter values are numerical. In addition to **pcb_get_mks()** and **pcb_set_mks()**, there are two more function pairs that are used to retrieve and store values of component parameters. The functions named **pcb_get_form_value()** and **pcb_set_form_value()** are used (respectively) to get and set values associated with constant formsets. For more information, refer to "Forms and Formsets" on page 4-11.

Similarly, string data can be retrieved and set using **pcb_get_string()** and **pcb_set_string()**. Arguments of these four functions exactly parallel those for **pcb_get_mks()** and **pcb_set_mks()** except that no scaling rules are applied. For more information on these functions, refer to chapter 10 of the "*AEL*" documentation. Functions are listed alphabetically in the AEL manual.

Callback Example - One Function per Component

Another way to program the above example makes use of the *clientData* field as a switch to select different logic within a single function. First, the item definition AEL code is revised as:

```
create_parm("A",                    // parameter
            "A parameter value",    // label
            68608,                  // attribute
            "StdFormSet",           // formset
            0,                      // unit code
            prm("StdForm", "10.0"), // default value
            list(dm_create_cb(      // callback list
                  PARM_MODIFIED_CB, // callback type
                  "abc_modified_cb", // callback function name
                  "A",              // clientData
                  TRUE))),          // callback enable
create_parm("B",                    // parameter
            "B parameter value",    // label
            68608,                  // attribute
            "StdFormSet",            // formset
            0,                       // unit code
            prm("StdForm", "10.0"),  // default value
            list(dm_create_cb(       // callback list
                  PARM_MODIFIED_CB,  // callback type
                  "abc_modified_cb", // callback function name
                  "B",               // clientData
                  TRUE))),           // callback enable
create_parm("C",                     // parameter
            "C parameter value",     // label
            68608,                   // attribute
            "StdFormSet",            // formset
            0,                       // unit code
            prm("StdForm", "10.0"),  // default value
            list(dm_create_cb(       // callback list
                  PARM_MODIFIED_CB,  // callback type
                  "abc_modified_cb", // callback function name
                  "C",               // clientData
                  TRUE))),           // callback enable
```

Next, the logic of the above three callback functions is combined into a single function. The value of the *clientData* is compared (*strcmp*) to *A*, *B* and *C* and the logic for *A* and *B* can be combined. This means if the value of either *A* or *B* is modified, the dependent parameter *C* is recalculated, but the code only needs to be provided once in the callback, as opposed to the previous example where it was provided separately for each parameter.

```
defun abc_modified_cb(cbP, clientData, callData)
{
  decl dependentParmData = NULL;

  if((strcmp(clientData, "A") == 0) ||
     (strcmp(clientData, "B") == 0))
    {
    decl a_mks = pcb_get_mks(callData, "A");
    decl b_mks = pcb_get_mks(callData, "B");
```

```
      dependentParmData = pcb_set_mks(dependentParmData,
                                                     "C", a_mks + b_mks);
      }
    else if(strcmp(clientData, "C") == 0)
      {
      decl c2_mks = pcb_get_mks(callData, "C")/2.0;

      dependentParmData = pcb_set_mks(dependentParmData, "A", c2_mks);
      dependentParmData = pcb_set_mks(dependentParmData, "B", c2_mks);
      }
    else
      fputs(stderr, "Illegal clientData value);

    return dependentParmData;
  }
```

Optimization Considerations

If any (or all) of the parameters of a component have associated modified parameter callback functions in the design environment AND these parameters are to be optimized, then the same relationships between the parameters must be included in the model code. That is, the design environment does not automatically enforce these relationships in the simulation environment during optimization (The relationships will, however be enforced in the display of components in a design when optimization variables are updated.) It may be desirable to restrict the optimizability of some parameters.

Developing and Testing Modified Parameter Callback Functions

As instructed in the tutorial, parameter callbacks should be included in the AEL file that holds the item definition for the associated element. In the tutorial, this file was called *mykit_item.ael*. The callbacks must be listed before the call to **create_item()** so they will be stored properly in the database of demand-loaded components. Use of demand-loaded components is optional, but it is recommended that you follow this format for your AEL files anyway, in case it is used in the future. For more information, refer to “Modifying the Item Definition File” on page 3-21, “Adding Demand Loaded Components” on page 3-29 and “Demand Loaded Components” on page 4-21.

Also, note that if the modified parameter callback function is being added to the AEL for a user-compiled model, this (AEL) file is subject to being overwritten by the design environment if any modifications are made therein. Keep a backup copy to put the callback information back in place as needed. You may need to reload the element AEL by entering **load("...")**; at the design environment command line (in the ADS Main window under **Options > Command Line**).

Some debugging information is available by entering the following line in the command line window:

debug_msg = 1;

Try this if you are not getting expected results. Debug messages are written to stderr. On unix, this is the window that ADS was started from. On PC, ADS must be started with a modified target in the shortcut, as described in “” on page 3-6.

Limitations of Parameter Callbacks

- Reference to data access components is not supported for elements that have parameters with modified parameter callback functions. A message is written to stderr if such a reference (PARM=file{...}) is made and debug_msg is enabled as described above.
- The modified parameter callback functions may be confused about unit scale factors when the right hand side of a parameter assignment involves an expression.
- Modified parameter callback functions do not modify dependent parameter values that are either variables or expressions.
- The AllParams parameter (that is used on model data items) is ignored.
- ADS component parameters which have been given the attribute of "not editable" (PARM_NOT_EDITED - attribute bit 1) are not changeable by modified parameter callback functions.
- There is currently no support for tuning, sweeping, optimization, back annotation, device operating point, etc.
- Parameter callbacks cannot use variable (VARs), they must be constants.

Netlist Callbacks

A parameter callback, described in the previous section, is a piece of AEL code that is executed when a parameter is modified on the schematic. A netlist callback is similar, but the AEL code for a netlist callback is executed when a schematic is netlisted for simulation. A netlist callback is defined for a component on the **create_item()** statement. Most components will not require a netlist callback. They can use the predefined netlisting rules for a general component or a component with a model, as explained in "Item Definition" on page 4-4. In a design kit, a netlist callback is typically used for the process include component, as defined in "Netlist Include or Process Component" on page 4-22.

A netlist callback is defined by the function **dm_create_cb()**. This function was defined above in "Adding a Callback to a Parameter Definition" on page 6-3. The only difference is that instead of PARM_MODIFIED_CB, the callback type for a netlist callback is ITEM_NETLIST_CB.

An example of a netlist callback is included in the "Adding a Netlist Include Component" on page 3-18. It is hard-coded to output the **#include** statement with the proper filename as determined from the design kit variables. The **#include** statement is described in "The #include Pre-processor Command" on page 4-29. It can also output the **#define** statement used to enable a specific corner case, also described in "The #ifdef and #define Pre-processor Commands" on page 4-30. The example from the tutorial is included here with more explanation.

```
create_item("mykit_include",

...

list (dm_create_cb (ITEM_NETLIST_CB,

"mykit_include_netlist_cb", NULL, TRUE)));
```

A part of the **create_item()** statement is shown above. Note that most of the arguments were omitted for this example. The callback information is a list, inserted before the list of component parameters. There are no parameters on this component. The list contains one or more calls to **dm_create_cb()**. The arguments to **dm_create_cb()** are:

- The first argument is the callback type. It must be ITEM_NETLIST_CB for a netlist callback.
- The second argument is the function name. This is the AEL code that will be executed during netlisting when this component is encountered in the schematic.
- The third argument is called client data. This is a string that may contain any information that you would like to use in the code. For this example, no client data is passed.
- The fourth argument is set to TRUE or FALSE and is used to enable or disable the callback.

The actual callback code is shown below.

```
defun mykit_include_netlist_cb (cbP, clientData, callData)
{
decl fileName="", netlistString="";

fileName = strcat(MYKIT_CIRCUIT_MODEL_DIR, "mykit_models.net");
netlistString=strcat(netlistString, "#include '", fileName,"'\n");

return(netlistString);
}
```

The arguments on the function are different than the arguments to **dm_create_cb()** when the callback was created. The callbacks on a callback of type ITEM_NETLIST_CB will always be the following:

- *cbP* - a pointer to the function, not needed for this example.
- *clientDat*a - a string defined in dm_create_cb() and passed in for reference. Also not needed for this example. The parameter callback example in "Callback Example - One Function per Component" gives a good example of how to use a clientData string.
- *callData* - also not needed for this example. Again, the parameter callback examples earlier in this chapter make use of these parameters.

For this example, the global path variable MYKIT_CIRCUIT_MODEL_DIR, which was set in *palette.ael*, is used to build the path to the model file. Then it is formatted into a string starting with the **#include** pre-processor statement. Finally the value *netlistString* is returned to the calling function. This is the string that is output to the netlist. A more complex example of a netlist callback, which includes the use of forms and formsets, is shown in "Example Process Component with Forms and Formsets" on page 4-25.

Layout vs. Schematic Comparison

The <*design_kit_name*>/netlist_exp directory contains all files needed by the ADS *Netlist Exporter*. The Netlist Exporter was designed for use in the ADS *Front End Design Flow*, which assumes that the layout was manually entered in a system outside of ADS. By providing rules files for each component in a design kit, the Netlist Exporter can output netlists in the proper form for many LVS tools, thus enabling a layout vs. schematic comparison to validate that the layout created outside of ADS matches the schematic created in ADS. Since the Netlist Exporter is configurable, rules can be written for any LVS tool. For detailed instructions on providing the rules files to insert into your design kit, refer to the ADS "*Netlist Exporter Setup*" documentation.

Creating Design Kit Documentation

All design documentation is saved in the *doc* subdirectory of the design kit directory. The documentation that you are recommended to provide in "The about.txt File" on page 4-36 and "Providing Basic Documentation" on page 3-13 is a very basic summary of the design kit which is intended to be presented in a standard design kit dialog in the future. More comprehensive documentation, such as detailed component information, can also be provided in the form of HTML files, which can be included by the end user into the ADS documentation set. At the present time, there is no automatic procedure for this. The only way to get documentation included in the full manual set is to incorporate it into the index file in the installation directory. This is not highly recommended, as any future releases of ADS may overwrite the index file that you have modified.

If you do wish to provide .html files with your design kit, you must instruct your customers to append the index file to the system index file, and to save a copy in case the master version gets overwritten. There are two tools in ADS to help you create this documentation. The ADS *"Electronic Notebook"* can help you generate the html document and the *"DesignGuide Developer Studio"* can help you generate the index file.

Layers and Preferences Files

The de/defaults directory of an ADS installation includes default preference (*.prf) and layers (*.lay) files for the schematic and layout windows. These files have names such as schematic.prf, schematic.lay, layout.prf and layout.lay. Additional preferences files are included for different schematic units, such as mm, mil and um. There are three configuration variables in *de.cfg* which are used by ADS to determine where the layers and preferences files should be read from. They are used as follows:

- PREFERENCES_DIR - When a new project is created in ADS, the default layers and preferences files are copied from this location.
- LAYERS_PATH, PREFERENCES_PATH - Each design file contains the names of the layers and preferences files used to create that design. A new design looks for the default files in the project directory. When a new or existing design is opened, if the project directory does not contain the specified layers or preferences files, the specified files are searched for in these directories.

Design kits may provide default layers and preferences files so that their components look the same way as they were created. However, the design kit infrastructure software does not provide any functionality to add the design kit directory to the path. This is because more than one design kit may contain conflicting definitions of layers and preferences and the user must have control over which they want to use. In a multi-chip design, the system cannot determine which layers to use for which parts.

This is something that a CAD manager or end user will need to control for his designs. Where internally developed design kits are used, a CAD manager can customize the environment to handle these files. A design kit prepared for distribution should contain instructions if the end user needs to make use of supplied default preference and layers files.

The design kit can handle this situation in one of two ways. First, a menu pick can be provided which will assist the user in making a conscious effort to set up these files. The callback from the menu pick would copy the custom versions of the files from the design kit directory to the project directory. Using the default names listed above (schematic.prf, schematic.lay, layout.prf, layout.lay) will ensure that each design uses these files, but if files with those names were already present in the project directory, any designs built with the default files will not look right after being replaced by the custom files. If a unique name is given to the custom files, any designs using them will have to be instructed to load the custom files by name. This can be done from the *Preferences for Schematic* and *Layer Editor* dialogs with the **Read** button.

In a controlled environment, custom AEL code can be added to a design kit which will modify the DIR or PATH configuration variables when a design kit is enabled so that the system always looks to the design kit directory for the custom layers and preferences files. This can be done by resetting the PREFERENCES_DIR path so that the custom files are copied from the design kit into a project when it is being created. The other option, setting the LAYERS_PATH and PREFERENCES_PATH variables to point first to the design kit directory will ensure that the custom files are used but will cause problems if a user tries to make changes in the layers or preferences dialog.

Advanced Topics

The topics in the remainder of this chapter are included to give you a preview of more of the capabilities that Advanced Design System has to offer. It is beyond the scope of this document to cover them in depth at this time. In some cases, references are given to other ADS documentation which will give you more information on the topics. You can also contact Agilent EEsof-EDA's Solution Services organization for assistance in implementing these features in your design kit.

Expressions

A design kit can include expressions for data processing before simulation or after simulation. A VAR component can be placed in schematic and expressions can be attached to it which will be evaluated before simulation. A MeasEqn component can be placed in schematic and expressions attached to it will operate on data generated during simulation. This type of expression can also be entered directly onto a data display.

There is no automatic procedure in the design kit infrastructure to tie the expressions in the expressions files in the design kit to the appropriate place in the schematic or data display windows, so they have to be copied manually, or a script provided for using them.

Templates

ADS supports two types of templates - simulation templates and data display templates. A simulation template can facilitate setting up common simulations and a data display template can contain a standard set of plots that can be used in different projects.

A design kit can include templates for simulation or for data display. Both tools in ADS have a menu pick Save As Template (data display) or Save Design As Template (schematic). Save a template that is designed specifically for use with your design kit and store it in the circuit/templates directory.

When a user loads the design kit, the appropriate path variable will be extended to include the design kit template directory and the templates will be available to users of your design kit. This path variable is DESIGN_KIT_TEMPLATE_BROWSER_PATH and it is referenced by HP_TEMPLATE_BROWSER_PATH in hpeesofbrowser.cfg. For more information on templates, refer to *"Using a Template"* in Chapter 2 of the ADS *"User's Guide"* and *"Using a Template in Your Display"* in Chapter 1 of the ADS *"Data Display"* manual.

Adding Custom AEL

There are times when a design kit developer may choose to include additional functionality in a design kit in the form of custom AEL code. The best way to load a custom AEL file is to have it loaded from the *boot.ael* file when a design kit is loaded. Other methods which might involve modifying configuration variables are not recommended. Your design kit should be easily distributable, so all files should reside in the design kit directory structure and no manual steps should be required of the end user.

Custom AEL files can be stored in the de/ael directory or the utilities directory. The location should be known by the *boot.ael* file so it can load the files with the full path.

To protect your AEL code, you can ship your design kit with the .atf files only. The .atf files are compiled versions of the AEL code. This will prevent users from modifying the code and the behavior of your kit. It should not be relied on as a form of security.

Adding Custom Menus to ADS

ADS has defined five user-definable menus in each window (main, schematic, layout) to which users may add their own menus picks. The ADS *"Customization and Configuration"* manual recommends one way to use these menus, but this is not recommended for design kits for the following reasons:

- It will overwrite the menu if it has been defined by another user or application.
- Another user or application may overwrite your menu.
- It requires adding files outside of the design kit directory structure.
- Trying to define user menus from within the design kit directory structure to avoid the previous problem requires loading an AEL file too late in the boot process to add menus to the main window so only schematic and layout menus can be added.
- Trying to load the AEL file earlier in the process to avoid the previous problem requires modifying a configuration variable (DESIGN_KIT_UI_AEL).
- Using the default function name (**app_add_user_menus()**) makes the function prone to being redefined by another user or application.
- Assigning a custom name to the function to avoid the previous problem requires modifying a configuration variable (USER_MENU_FUNCTION_LIST).

A distributable design kit should not modify files outside of the design kit structure and should not modify any configuration variables in saved configuration files.

For design kits that are for internal use only, the following are some methods to use to add custom menus.

1. Create a new file $HOME/hpeesof/de/ael/usermenu.ael and copy the code from Table 6-1 into the file. This controls adding the menus in each window, Main, Schematic and Layout. The *usermenu.ael* file is read automatically when ADS is started, and the function **app_add_user_menus()** is automatically called with the appropriate winType when each window is being created. Notice the custom

main menu 1 item that now appears in the ADS Main window shown in Figure 6-1.

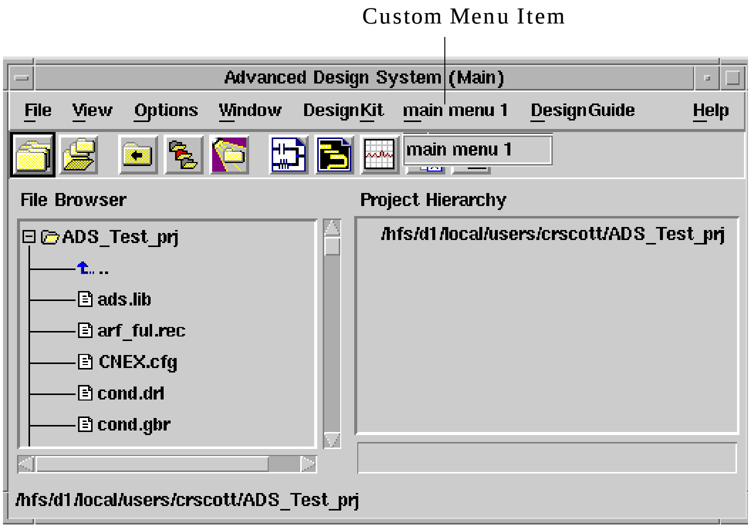


Figure 6-1. Custom User Menu

2. Following the sample shown for the Main window, add appropriate function calls to **api_add_menu()** for each window type. Also add the callback code. Save the file.

3. If you want to change the function name from **app_add_user_menus()** to a name unique for your kit, save a new variable USER_MENU_FUNCTION_LIST in $HOME/hpeesof/config/de_sim.cfg but don't forget to copy the old contents of that list since the local definition will override the system one, which is in $HPEESOF_DIR/config/de.cfg. Also check $HPEESOF_DIR/custom/config for any files that redefine that variable. The new definition might look like this:

```
USER_MENU_FUNCTION_LIST = app_add_user_menu:mykit_add_user_menu
```

   Beware that these menus are not protected and may be overwritten by an application or user that is not using code like that in Table 6-1 that checks for a free slot.

Table 6-1. ADS Window Menu Control

```
//winType = MAIN_WINDOW, SCHEMATIC_WINDOW or LAYOUT_WINDOW
defun mykit_find_empty_user_menu( winType )
{
    decl menuCascadeH=NULL,i;
    // Do not edit the elements of the list mykitUserMenuList.
    decl mykitUserMenuList = list(deUserMenuName, deUser2MenuName,
                   deUser3MenuName, deUser4MenuName, deUser5MenuName);
    api_select_window(winType);
    for (i = 0; i < listlen(mykitUserMenuList); i++)
    {
        menuCascadeH = api_find_menu(NULL, mykitUserMenuList[i]);
        if (menuCascadeH != NULL && api_total_sub_menus(menuCascadeH) <= 0)
          return(menuCascadeH);
        menuCascadeH = NULL;
    }
    return(NULL);
}
// winType = MAIN_WINDOW, SCHEMATIC_WINDOW or LAYOUT_WINDOW
defun app_add_user_menus(winType)
{
    decl menuCascadeH, name;
    decl menuPickName, menuCB;
    menuCascadeH = mykit_find_empty_user_menu(winType);
    if(menuCascadeH == NULL)
        return;
    if(winType == MAIN_WINDOW)
    {
        api_set_menu_label(menuCascadeH, "main menu 1");
        api_add_menu(menuCascadeH, api_create_menu("main menu 1", NULL,
"main_menu_cb1", NULL, NULL, NULL));
    }
    else if(winType == SCHEMATIC_WINDOW)
    {
          // Add code for schematic window menu item here.
    }
    else if (winType == LAYOUT_WINDOW)
    {
          // Add code for layout window menu item here.
    }
}
```

Adding Custom Models to the ADS Simulator

Sometimes a model used in a design kit will not be available in the ADS simulator. This type of model will have been created by your company as a *user-compiled model*. In this case, your design kit must include a custom version of the simulator that includes this model, and it will only be compatible with the specific release of ADS for which it was created. For more information on how to create a user-compiled model, refer to the ADS *"Analog/RF User-defined Models"* documentation.

As described in "Overview of the File Structure" on page 2-1, the user-compiled simulator will need to be provided for all platforms. Some examples of the names of directories in which to place each executable are listed below.

bin/hpux10

bin/hpux11

bin/aix

bin/sunos5.6

bin/winnt

If you supply a custom executable, your design kit will have to include directions telling your customer to modify their search path so that the new *hpeesofsim.exe* is picked up before the built-in version. Your user will also have to be informed that if they have a design kit from another vendor which also contains a custom executable, they will not be able to run a simulation with parts from both kits.

This process will become easier in the near future when user-compiled models can be delivered in the form of a dynamically linked library. The multiple design kit limitation will also be resolved.

ADS Layout Files

The ADS design kit infrastructure software is aimed initially at design kits for use with the ADS *Front End Design Flow*. This design flow assumes that a schematic is created and simulated in ADS and then the layout is entered manually in another system for layout post-processing. Since there are many users of the ADS native layout tool, the standard design kit structure also provides some standard directories for these needs, which are documented below.

There is no special code in the design kit infrastructure code that uses these directories. Custom AEL or custom menu picks can be provided with a design kit or by a local CAD manager to help users include these files in their design work.

circuit/artwork	-	This directory is provided for AEL artwork macro files for ADS layout.
circuit/substrates	-	This directory is provided for Momentum substrate files.
de/defaults	-	Custom layers and preferences files for layout can be put here. For more information, refer to "Layers and Preferences Files" on page 6-17.
drc	-	Use this directory for files needed by the ADS Design Rule Checker tool.

Chapter 7: Standardizing Existing ADS Design Kits

Most design kits in use with ADS today were created before the standard design kit structure was developed. This chapter describes the process of standardizing an existing ADS design kit.

Design Kit Parts

To update an existing ADS design kit to conform to the new standard structure, first study Chapter 2, Understanding the ADS Design Kit File Structure and Chapter 4, Basic Parts of an ADS Design Kit to understand the new structure. Then determine which of the directories in the new structure are best suited to hold the data from your old design kit. Note which directories and files are required. Create the new directory structure and copy your old files into the new directories.

If you did not use one before, this is a good time to start using a revision control or history management system, which will tag each file with a version each time it is updated. The overall set of files can also be tagged at a release, enabling you to trace individual files back to their actual kit version.

From Chapter 2, Understanding the ADS Design Kit File Structure, you will learn which are the required parts of a design kit. Chapter 4, Basic Parts of an ADS Design Kit will give more details on each. Especially make sure that the template *ads.lib* file is created. This is often missing from older design kits. For more information on the ads.lib file, refer to "The ads.lib Template" on page 4-33.

Naming Convention

Make sure that all names used in your design kit follow the naming guidelines outlined in Chapter 4, Basic Parts of an ADS Design Kit. Since ADS does not currently have a true hierarchical level called a library, all components are stored in the same name space when they are entered into the system. If components from different libraries have the same name, the last one loaded will overwrite any that were loaded before it. To ensure that your components do not collide with components from the built-in libraries or from other design kits, prefix each component name with a unique identifier such as the name of the design kit. This also applies to all global variables, subcircuit names in external files, as well as external data files names.

Component Selection Method

Next, you should decide if the current method of selection of components is adequate. For a discussion of the pros and cons of each component selection method, refer to "Component Palette vs. Library Browser" on page 4-15. The components in your kit can be made available from the component palette or the library browser or both. You can even have some available one way and others available the other way.

Palette Bitmaps

One part of old design kits that usually varied from kit to kit was the name of bitmap directories. The standard calls for bitmaps to be stored in the circuit/bitmaps directory under the *pc* or *unix* subdirectories. The name of the bitmap can be the same since they are stored in different directories. The section on "Custom AEL Code" on page 7-2 lists another bitmap change that may be required.

You may also choose to update your bitmap graphics at this time if it does not include any indication of the design kit it came from. The "*DesignGuide Developer Studio*", which is available with your ADS software, has a tool to help create bitmaps specifically for ADS. For more information on bitmaps, refer to "Bitmaps" on page 4-17 and "Creating a Component Palette and Bitmaps" on page 3-14.

Custom AEL Code

Some old design kits include modified versions of the old infrastructure software. Use of this old code should be discontinued as it causes new design kits to be unusable. Custom AEL code can still be provided, and there are some cases where it must be provided, but you should update your AEL code to work with the new design kit infrastructure code, the code in ADS that loads design kits.

Since custom menus are not provided with the design kit infrastructure at this time, that is one area for which you must continue to provide custom code. It is advisable that you make sure every function that you write in AEL has a unique name to avoid collision with other AEL functions in the ADS system or in other design kits. All function names in your design kit should begin with a unique identifier such as the name of your design kit.

Some design kits developed previously used the AEL constant DKBITMAPSTRING in calls to **create_item()**. This constant was set to bitmaps or pcbitmaps, depending on the platform. Since the correct directories for bitmaps are now bitmaps/pc and bitmaps/unix, the DKBITMAPSTRING constant should no longer be used. It will remain in the backward compatibility code so old kits continue to work.

Design Kit Distribution

After you have modified your design kit to conform to the standard structure, and have assigned names which include a unique identifier to all parts of the design kit, review Chapter 5, Completing the Design Kit to understand the recommended procedures for packaging, distribution and support of your design kit. Distributing your design kit as a simple .zip file is recommended. This will ensure that it works with the ADS Design Kit installation code. Complex installation scripts are not recommended as they have caused problems in the past which can easily be avoided.

Kits that do not conform to the Standard

Finally, if you have found that some parts of your design kit just do not conform to the standard structure, please contact the *Agilent EEsof-EDA Design Kit project team*. The team can review your needs and recommend the best way to solve your problem. The standard design kit structure may need to be extended to cover your needs.

Chapter 8: Setting Up Design Kit Software and Menus

This chapter describes the details of configuration files and variables which are used to enable and disable the common design kit software. This includes software for loading design kits, as well as the software to control the menus and dialog boxes that are shipped with ADS for use with all design kits. This information is included for advanced users or users of old design kits who experience problems between the new software and an old design kit.

Configuration Files

Each individual Advanced Design System tool has its own unique configuration file. These configuration files contain information required by the software to install and enable the tools.

If you are an advanced design kit user, there are two configuration files that you should be aware of:

- The *de_sim.cfg* file.   This is the general ADS configuration file.
- The *design_kit.cfg* file.   This is the specific design kit configuration file.

The configuration files in Advanced Design System are read from three locations:

- $HPEESOF_DIR/config   This is the default location where configuration files are located when shipped with the software.
- $HPEESOF_DIR/custom/config   This location is reserved for customization, typically by CAD managers.
- $HOME/hpeesof/config   This location can be used for customization by the end user. Configuration files in this location may also be written to by the software.

Note   Any customization set up in $HPEESOF_DIR (software installation directory) should be confined to the *custom* subdirectory. This is because anything outside of the *custom* subdirectory may be overwritten by a subsequent update to the software.

The *de_sim.cfg* file is read from all three locations in the order listed above. If a variable is defined in more than one location, the last value read is the one that will be recognized by the system.

The *design_kit.cfg* file is included by adding the following line to the *de_sim.cfg* file:

```
+ design_kit
```

When **+ design_kit** is encountered in a de_sim.cfg file, the design_kit.cfg file is only read from one location, as opposed to the three locations described above. This single location is the same directory that contains the de_sim.cfg file currently being read.

Current Configuration Variables

The information described in Table 8-1 is a list of configuration variables used by the current design kit infrastructure software beginning with ADS2001. This is the general code that installs and enables design kits, as well as the user interface code such as menus and dialog boxes common to all design kits. This code should be differentiated from any software or code that is specific to a particular design kit. Any code referred to in this section is application extension language (AEL) code. For more information on AEL code, refer to the Advanced Design System "*AEL*" manual.

If you are a general user of new ADS design kits, it is not important to understand the information provided in this section. This information is provided for CAD administrators, design kit developers or anyone who has worked with design kits in ADS prior to ADS2001, when the design kit infrastructure code had to be manually configured. These variables are now all set automatically in ADS.

Anyone who used design kits prior to ADS2001 should check their *de_sim.cfg* files in all user configurable locations and remove (or preferably comment out with a # sign in column one) any configuration variables used for general design kit configuration in the past. Refer to "Accessing Old Design Kit Software" on page 8-5 for a review of those variables. The last section, "Accessing Both Old and New Design Kit Menus Simultaneously" on page 8-6, will help you reset the necessary variables if you have an ongoing need for access to the old design kit software.

A description of each design kit variable required for current ADS design kit infrastructure software is provided in Table 8-1.

Table 8-1. ADS Design Kit Variables

DK_AEL_PATH This variable is set in the $HPEESOF_DIR/config/design_kit.cfg file. The DK_AEL_PATH variable is set to the directory where the common design kit AEL code resides. This variable is used by the next few variables in the file. DK_AEL_PATH={$HPEESOF_DIR}/design_kit/ael
Note: The next three variables described in this table are referenced in $HPEESOF_DIR/config/de.cfg when ADS boots up.
DESIGN_KIT_UI_AEL This is the AEL code related to the Design Kit menu in the Main ADS window. It is read very early during boot-up. DESIGN_KIT_UI_AEL= {%DK_AEL_PATH}/dk_menu
DESIGN_KIT_STARTUP_AEL This is a list of AEL files read later during boot-up. dk_startup contains code to load the specific design kits which are currently configured. The other files contain code related to the common design kit menus and dialog boxes. DESIGN_KIT_STARTUP_AEL={%DK_AEL_PATH}/dk_startup:{%DK_AEL_PATH}/dk_explorer
DESIGN_KIT_PROJECT_AEL This AEL file contains code that is invoked every time a user opens a project. DESIGN_KIT_PROJECT_AEL={%DK_AEL_PATH}/dk_project_attach
DESIGN_KIT_BROWSER_PATH This variable is used to extend the search path for the library browser. The common design kit software automatically appends “<design_kit_path>/circuit/records” to this path whenever a design kit is loaded. All the control files and the .idf files in these directories are read by the library browser to create the Analog/RF related libraries/sub-libraries. The library browser is a separate executable program, which is closed and restarted to pick up the new path. This variable is referenced by HPANALOGRF_BROWSER_PATH in $HPEESOF_DIR/config/hpeesofbrowser.cfg. DESIGN_KIT_BROWSER_PATH=

Table 8-1. ADS Design Kit Variables

DESIGN_KIT_TEMPLATE_BROWSER_PATH This variable is used to extend the search path for the template browser. The common design kit software automatically appends “<design_kit_name>|<design_kit_path>/circuit/templates” to this path whenever a design kit is loaded. Each entry is in the form of <name>|<path> where <name> is the top level node displayed in the browser and <path> is the path to the directory containing the templates. The template browser is a separate executable program, which is closed and restarted to pick up the new path. This variable is referenced by HP_TEMPLATE_BROWSER_PATH in $HPEESOF_DIR/config/hpeesofbrowser.cfg. DESIGN_KIT_TEMPLATE_BROWSER_PATH=
DESIGN_KIT_NO_MENU Obsolete variable used by design kit code prior to ADS 2001. Design kit menus are now available full-time for all users. DESIGN_KIT_NO_MENU=TRUE

Table 8-1. ADS Design Kit Variables

DESIGN_KIT_LOCAL_OVERRIDE Obsolete variable used by design kit code prior to ADS 2001 to suppress reading of the design kit configuration file ads.lib from system-wide location. This functionality is now provided in a more flexible manner from the design kit software and the configuration variable DESIGN_KIT_LEVELS_ENABLED. DESIGN_KIT_LOCAL_OVERRIDE=FALSE
DESIGN_KIT_LEVELS_ENABLED This variable refers to the ads.lib files that will be read to load design kits. This is similar to the obsolete variable DESIGN_KIT_LOCAL_OVERRIDE, but is less ambiguous and gives the user more flexibility and control. There is a design kit menu pick that opens a dialog to control this variable. To manually change it to suppress the reading of the ads.lib file at the specified level, remove the name of the level from the list. Order of the list is not important and colons do not need to be retained. PROJECT:SITE is a valid value instructing the system to read ads.lib from the current project and the site locations only. Locations are as follows: SITE: $HPEESOF_DIR/custom/design_kit/ads.lib USER: $HOME/hpeesof/design_kit/ads.lib STARTUP: The directory from which ADS is invoked on unix. This can be set as a short-cut property for PC systems. PROJECT: If the project level is enabled, the ads.lib file will be read if available any time a project is opened, and design kits specified in that ads.lib file will be loaded. Note that leaving a project will not unload any design kits as the ADS system does not currently have the capability to remove the system components once they are loaded. For more information on the ads.lib file, refer to "The ads.lib Template" on page 4-33. DESIGN_KIT_LEVELS_ENABLED=SITE:USER:STARTUP:PROJECT

Accessing Old Design Kit Software

Prior to ADS 2001, to enable a design kit and the design kit menus, the following configuration variables had to be set manually in de_sim.cfg at the local ($HOME/hpeesof) or custom ($HPEESOF_DIR/custom/config) level.

To load the AEL code needed to load design kits and the menus:

```
DESIGN_KIT_DIRECTORY={$HPEESOF_DIR}/design_kit

USER_AEL={%DESIGN_KIT_DIRECTORY}/design_kit_startup
```

LOCAL_AEL={%DESIGN_KIT_DIRECTORY}/design_kit_project_attach

To suppress reading the ads.lib design kit file from the system level:

DESIGN_KIT_LOCAL_OVERRIDE=FALSE Set it to TRUE to only read local ads.lib files.

To turn on the design kit menus in all ADS windows:

DESIGN_KIT_NO_MENU=FALSE By default this is set to TRUE so design kit menus are not visible unless needed.

To enable design kit menus in schematic and layout windows:

USER_MENU_FUNCTION_LIST=app_add_design_kit_menus This is only needed for the rare design kits which supply a custom schematic or layout menu. It works with the DESIGN_KIT_NO_MENU variable, which ultimately controls the menu visibility. Set DESIGN_KIT_NO_MENU=FALSE to see the menus in schematic or layout windows.

With the release of ADS 2001, the first three of these were moved into the system design_kit.cfg and automatically loaded. The next two were defaulted in the system design_kit.cfg and only needed to be set in a local or custom de_sim.cfg if values other than the default were desired. The last one was not defined in design_kit.cfg. It still needs to be set manually if required.

Accessing Both Old and New Design Kit Menus Simultaneously

In an Add-on release to ADS 2001, new design kit software is provided. The old functionality was retained where needed to support existing design kits, but completely new menus are provided and turned on automatically without user intervention.

With the new configuration variables, the old design kit menu in the ADS *Main* window is no longer readily accessible. All design kit related activities should be performed through the new menus. The only reason to access the old menu is for other utilities that were historically on the same menu. If access to the old menu is required, the following steps should be taken:

1. Remove or comment out general design kit configuration variables (listed in this chapter) from any user configurable location. These will interfere with the new software. If a design kit configuration variable is not discussed in this

chapter but is in your system configuration, perhaps it applies to a specific design kit. These can remain as they are.

2. After removing old variables, restart ADS and verify that only the new design kit menu is visible. After this has been verified, close ADS. [need a link to a screen dump so user knows what new menu looks like]

   To gain access to the old software without disabling the new software, perform the following steps:

3. Copy $HPEESOF_DIR/config/design_kit_old.cfg.unix
   or
   $HPEESOF_DIR/config/design_kit_old.cfg.win32
   to
   $HOME/hpeesof/config/design_kit_old.cfg
4. Add the following line to $HOME/hpeesof/config/de_sim.cfg

   ```
   + design_kit_old.cfg
   ```
5. Follow the directions in the design_kit_old.cfg file or read the information below and modify design_kit_old.cfg as needed.

Remember that the value of a variable in $HOME/hpeesof/config will take precedence over one set in $HPEESOF_DIR/custom/config or $HPEESOF_DIR/config (see "Configuration Files" on page 8-1). Choose **Options > Configuration Explorer** from the the ADS *Main* window to check for conflicts. Other tools may have set these variables to different values. Resetting them here may disable other tools. This can be fixed by combining all needed values together, separated by a colon (:).

Example:

```
USER_MENU_FUNCTION_LIST=app_add_user_menus:app_add_design_kit_menus
```

Note To enable better comparisons within the Configuration Explorer, set CONFIG_EXPLORER_CMP_VARS in de_sim.cfg, but beware that this makes the Explorer run very slow on networked systems. It works best on a stand-alone PC installation.

To make the old design kit menu visible in the main ADS window, change this value to FALSE and restart ADS. You may need to iconize and restore the window once if the menu pick isn't visible immediately.

DESIGN_KIT_NO_MENU=TRUE

These reset the USER_AEL and LOCAL_AEL path to load the software. Definitely check for conflicts with these AEL variables.

DESIGN_KIT_OLD_DIRECTORY={$HPEESOF_DIR}/design_kit

USER_AEL={%DESIGN_KIT_OLD_DIRECTORY}/design_kit_startup

LOCAL_AEL={%DESIGN_KIT_OLD_DIRECTORY}/design_kit_project_attach

Uncomment this variable to turn on the old design kit menu in schematic or layout windows.

USER_MENU_FUNCTION_LIST=app_add_design_kit_menus

This is an obsolete variable that formerly controlled the scope of ads.lib files. For information on using the new method of controlling the scope of the ads.lib files, refer to DESIGN_KIT_LEVELS under "Current Configuration Variables" on page 8-2.

DESIGN_KIT_LOCAL_OVERRIDE=TRUE

Viewing Configuration Files and Variables

A useful tool for viewing configuration files and variables within Advanced Design System is the *ADS Configuration Explorer*. To launch the Configuration Explorer:

From the ADS *Main* window, choose **Options > Configuration Explorer**.
The ADS Configuration Explorer dialog box appears.

For more information on the ADS Configuration Explorer, refer to "*Viewing Details of the Current Configuration*" in Chapter 1 of the ADS "*Customization and Configuration*" manual.

For more information on configuration files and configuration variables in ADS, refer to Chapter 1 of the ADS "*Customization and Configuration*" manual and Chapter 3 of the "*AEL*" manual.

Disabling the Design Kit Software

The new ADS design kit software is designed to be compatible with the old design kit software as it was shipped from the factory. However, there are some extreme cases where an old design kit created outside of the factory may not operate with the new software loaded. This is apparently due to the fact that the old factory software was modified for a customer or by a customer, without making the names of the AEL functions unique. Since all AEL functions are global in scope, any custom AEL code has the potential to overwrite an existing AEL function, if the custom code does not have a unique prefix on all functions and global variables.

Any design kits with the problem described above will also not be able to co-exist with other design kits which follow the design kit standard structure and use the standard method to enable the design kit. These design kits should be upgraded to follow the standard structure and use the standard software. Until they are upgraded, the following steps can be taken to disable the new design kit software. This can be done at the user or system level.

1. Copy $HPEESOF_DIR/config/design_kit.cfg to $HPEESOF_DIR/custom/config or $HOME/hpeesof/config.
2. Edit *design_kit.cfg* file as follows to undefine the variables that load the design kit AEL files.

   DESIGN_KIT_UI_AEL=

   DESIGN_KIT_STARTUP_AEL=

   DESIGN_KIT_PROJECT_AEL=
3. Add **+design_kit** anywhere in the de_sim.cfg file which resides in the same directory where you edited design_kit.cfg. If the de_sim.cfg file doesn’t exist, create a new file.

Appendix A: ADS Design Kit Development for RFIC Dynamic Link

This appendix describes general information related to design kits created for use in the RFIC Dynamic Link Flow between Advanced Design System and Cadence DFII.

RFIC Dynamic Link

The RFIC Dynamic Link is an EDA framework integration product based on Inter-Process Communication (IPC), rather than data file translation, maximizing data integrity and ease of use. The Dynamic Link enables you to create a design in the Cadence EDA framework and then simulate your design in Advanced Design System. This provides you with the powerful capabilities of both EDA design environments.

The RFIC Dynamic Link documentation set provides fundamental information on the installation, usage and customization of your EDA environment. Ensure that the Dynamic Link is properly configured before attempting to use the product.

Design Kits for RFIC Dynamic Link

ADS foundry models have been developed to provide access to a specific process within Advanced Design System when using RFIC Dynamic Link. To give the simulator access to these models, the RFIC Dynamic Link documentation recommends that you place a generic include component called *NetlistInclude*. For more information on the *NetlistInclude* component, refer to "Netlist Include or Process Component" on page 4-22. On the include component, you must enter the name and path of model files to be referenced, as well as specifying any corner case identifiers.

When developing a design kit for use in the RFIC Dynamic Link flow, it is recommended that you create a custom process component instead of requiring the end user to configure the *NetlistInclude* component. A custom component can contain the names of the model files so the end user does not need to know this information. It can also present the end user a simple list to select the appropriate corner case based on meaningful criteria. This process component, coupled with reusable models or data files, can then be used as the basis for a design kit for use in the ADS *Front-End Design Flow*.

For an example of this type of process component, refer to "Example Process Component with Forms and Formsets" on page 4-25.

Appendix B: ADS Design Kit Development for IFF

This appendix describes general information related to creating design kits for design flows involving Advanced Design System and the Intermediate File Format (IFF).

Intermediate File Format

Intermediate File Format (IFF) is an ASCII file format that is both platform and application independent. The file has a simple, line-oriented command structure with a fairly rich set of constructs, thus simplifying design transfer. The IFF translators offered by Agilent Technologies provide a means for transferring IFF files between Advanced Design System and third-party electronic design automation (EDA) tools such as Cadence Design Framework II or Boardstation from Mentor Graphics Corporation.

Foundry kits developed for use in design flows involving ADS and another EDA tool, where designs are transferred via IFF, should follow the standard structure (as described in this document) for files used on the ADS side of the link.

For more information on using the Intermediate File Format in ADS, refer to the "*Understanding Component Library Requirements*" in chapter 2 of the "*IFF Schematic Translation for Cadence*" documentation.

For more information on creating design kits for use in this combined environment, contact Agilent EEsof-EDA Solution Services.

Index