Agilent Technologies

Advanced Design System 2002

Netlist Exporter Setup

February 2002

Notice

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

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

Warranty

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

Restricted Rights Legend

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

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

Copyright © 2002, Agilent Technologies. All Rights Reserved.

Acknowledgments

Cadence® and Analog Artist® are registered trademarks of Cadence Design Systems Incorporated.
Design Framework II™ and Composer™ are trademarks of Cadence Design Systems Incorporated.
Copyright © 2001 Cadence Design Systems Incorporated. All rights reserved.

Mentor Graphics® is a registered trademark of Mentor Graphics Corporation.

Copyright © 1997-2001 Mentor Graphics Incorporated. All rights reserved.

Contents

Chapter 1: Setup

This chapter covers the installation configuration file settings for Front End Flow. For information on using Front End Flow, refer to the ADS *Netlist Exporter* manual.

License Requirements

The following license is required for Front End Flow to operate in Advanced Design System:

- *Spice_netlist_trans*

This license is associated with the E8880 SPICE Netlist Translator module.

Note Before continuing, ensure that you have a valid license for the ADS schematic environment. For more information on ADS licenses, refer to "*Setting up Licenses on UNIX Systems*" in the ADS "*Installation on UNIX Systems*" manual or "*Setting Up Licenses on PC Systems*" in the ADS "*Installation on PC Systems*" manual.

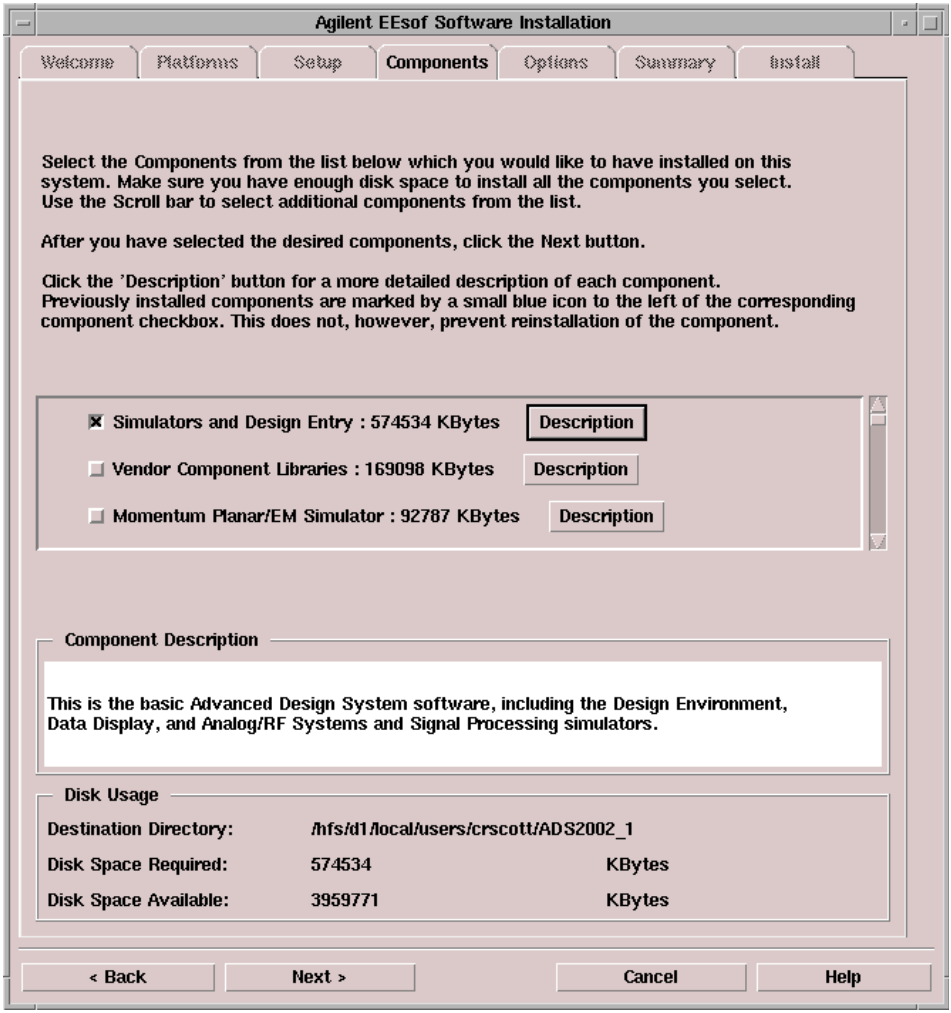
Installing Netlist Exporter

The Netlist Exporter installation procedure continues to be improved to make it easier for you to install and configure. Netlist Exporter is now installed with each installation of Advanced Design System that includes the *Simulators and Design Entry* component. For more detailed information on the Netlist Exporter installation procedure, refer to the information below.

To install Netlist Exporter:

1. For a UNIX installation, follow the instructions in the ADS "*Installation on UNIX Systems*" manual to run the SETUP utility and load the *install* program.

   For a PC installation, follow the instructions in the ADS "*Installation on PC Systems*" manual. The setup program will automatically bring up the Software Installation Wizard.



2. After the *Agilent EEsof Installation Manager* starts, you are prompted to select one of the following installation options:

   - **Typical** - If you choose a *Typical* installation, the Netlist Exporter will be automatically installed.
   - **Complete** - If you choose a *Complete* installation, the Netlist Exporter will be automatically installed.

- **Custom** - If you choose a *Custom* installation, you must select the *Simulators and Design Entry* component from the scroll-down list in the *Agilent EEsof Software Installation* dialog box.

Note The *Simulators and Design Entry* component is the basic ADS software, including the Design Environment, Data Display, and Analog/RF Systems and Signal Processing simulators. This is a minimum requirement for Netlist Exporter.

---

3. Continue the installation process by following the setup instructions. After the installation is complete, you will have the following:
   - In *$HPEESOF_DIR*, there will be a *netlist_exp* directory. This is the installation home of Front End Flow.
   - In the *config* directory, there will be a new *CNEX.cfg* file, which contains the default settings for the Front End Flow.
   - A menu labeled *Netlist Export* will appear in the tools menu on Schematic windows.

For more information on installation procedures, refer to the ADS *"Installation on UNIX Systems"* or ADS *"Installation on PC Systems"* manual.

Configuration File Settings

The following configuration options exist and can be modified in *CNEX.cfg* files. Modifications can be made to the following CNEX.cfg files:

- $HPEESOF_DIR/config/CNEX.cfg
- $HPEESOF_DIR/custom/config/CNEX.cfg
- $HOME/hpeesof/config/CNEX.cfg.

Note Do not make modifications to the file *CNEX.cfg* that is within your working project. This file is automatically updated by the Front End Flow application.

CNEX_TOOL

This value is used to construct AEL paths and component paths so that appropriate code and component definitions will be used. The *CNEX_TOOL* value will default to Assura after installation. The netlist export and component dialog settings modify the *CNEX_TOOL* value in the *CNEX.cfg* file that is within the current working directory. No manual change is necessary.

CNEX_HOME_DIR

This value specifies the home directory for user defined AEL customizations and component definitions. It is available as a shorthand notation for the *CNEX_EXPORT_FILE_PATH* and *CNEX_COMPONENT_PATH* variables.

Note It is recommended that this value not be changed from its default value of *{$HOME}/hpeesof/netlist_exp*. If you do *not* wish to have user customizations available, remove *CNEX_HOME_DIR* from *CNEX_EXPORT_FILE_PATH* and *CNEX_COMPONENT_PATH*.

CNEX_CUSTOM_DIR

This value specifies the directory used for site-wide Front End Flow customizations. The default value is *{$HPEESOF_DIR}/custom/netlist_exp*. If you do not wish to follow the ADS standard for site wide customization, this directory can be changed into any Unix or PC path.

CNEX_INSTALL_DIR

This value specifies the installation point of the Front End Flow software. The default value is *{$HPEESOF_DIR}/netlist_exp*. If you wish to maintain multiple versions of Front End Flow software, you can install the Front End Flow application at locations outside of the *HPEESOF_DIR* directory tree, and alter *CNEX_INSTALL_DIR* to point to that directory.

Note *CNEX_INSTALL_DIR* must always be set to a valid Front End Flow installation. The default un-customized files in *CNEX_INSTALL_DIR* will be loaded each time during netlist exporting, even if other customization files that contain the default functions have been created.

CNEX_DESIGN_KIT_PATH

This value will be set during netlist exporting, and will update to include all of the component directories that are available for *CNEX_TOOL* in the design kits that are loaded in the ADS session. No manual change is necessary.

Note This value is empty in the default *CNEX.cfg* file–do not change this value.

CNEX_DESIGN_KIT_AEL_PATH

This value is set during netlist exporting, and will update to include all of the custom AEL code for *CNEX_TOOL* that is available for a design kit. No manual change is necessary.

Note This value is empty in the default *CNEX.cfg* file–do not change this value.

CNEX_EXPORT_FILE_PATH

This value specifies the directory search order for AEL code that will be loaded during netlist exporting. The Front End Flow netlister will always load the files *cnexGlobals.ael* and *cnexNetlistFunctions.ael* when a netlist is to be generated. The file loader will load **each** *cnexGlobals* and *cnexNetlistFunctions* file found within *CNEX_EXPORT_FILE_PATH*.

Note When an AEL file is loaded, it will override existing variables and functions. By loading the files in the order specified, it is possible for later files to override the default functions that are shipped for Front End Flow. Therefore, place the paths in priority in the path list: lowest priority path first in the path list and highest priority path at the end of the list.

The default *CNEX_EXPORT_FILE_PATH* value is the following:

{%CNEX_INSTALL_DIR}/ael;{%CNEX_INSTALL_DIR}/ael/{%CNEX_TOOL};{%CNEX_CUSTOM_DIR}/ael;{%CNEX_CUSTOM_DIR}/ael/{%CNEX_TOOL};{%CNEX_HOME_DIR}/ael;{%CNEX_HOME_DIR}/ael/{%CNEX_TOOL};{%CNEX_DESIGN_KIT_AEL_PATH}

Note The *CNEX_EXPORT_FILE_PATH* value must be a single line in the configuration file.

CNEX_COMPONENT_PATH

This value specifies the directory search order for Front End Flow component definitions.

Note *Only* the first definition encountered in the *CNEX_COMPONENT_PATH* will be read. Place the paths in priority in the path list: highest priority path first in the path list and lowest priority path at the end of the list.

The default value after installation is the following:

{%CNEX_DESIGN_KIT_PATH};{%CNEX_HOME_DIR}/components/{%CNEX_TOOL};{%CNEX_CUSTOM_DIR}/components/{%CNEX_TOOL};{%CNEX_INSTALL_DIR}/components/{%CNEX_TOOL}

Note The *CNEX_EXPORT_FILE_PATH* value must be a single line in the configuration file.

Design Tool Support

The following design tools are supported by ADS Front End Flow:

- Cadence Dracula
- Cadence Assura
- Mentor Graphics* Calibre

Component Support

These tools have component definitions available for the ADS standard parts in the *$HPEESOF_DIR/netlist_exp/components* directory. User defined libraries and parts require Front End Flow customization. Refer to Chapter 3, Component Definitions for customization information. Optionally, you can contact the Agilent Technologies Solution Services group to contract special support for your user defined libraries and parts.

Netlist Options Support

Custom AEL code to support netlist options for the supported design tools is provided in *$HPEESOF_DIR/netlist_exp/ael*.

Unsupported Design Tools

To use a non-supported design tool with ADS, you will need to customize Front End Flow to work with that tool. Refer to "Adding Tools to Front End Flow" on page 1-9 for customization information. Optionally, you can contact the Agilent Technologies Solution Services group to contract special support for your unsupported design tool.

Front End Flow Directory Structure

Front End Flow has many layered elements. Each subsequent layer adds new functionality to the product. Part of the layering is the Front End Flow directory structure.

The following four subdirectories are created in the Front End Flow directory wherever a netlist_exp is appropriate:

ael	The *ael* directory contains the compiled AEL (atf) program files with or without the associated AEL files. Each directory specified in *CNEX_EXPORT_FILE_PATH* will be searched for AEL files relevant to Front End Flow (see "CNEX_EXPORT_FILE_PATH" on page 2-6). The relevant files will be loaded. For more information on AEL files as they relate to Front End Flow, see Chapter 4, Customizing a Netlister, and Chapter 5, Setting up GUI Options. Within the *ael* directory, there can be subdirectories that contain AEL files that are specific to a particular tool. These subdirectories will normally be in *CNEX_EXPORT_FILE_PATH*, so that appropriate files will be loaded and override the default capability of the exporter for a particular tool.
components	The *components* directory contains subdirectories for the tools supported by Front End Flow. In each tool subdirectory, there are component definition files for the components set up for Front End Flow for that particular tool. See Chapter 3, Component Definitions for information on component definition files. **Note** Each tool supported for Front End Flow must have a separate directory for its component definitions. There is no facility to share a component definition between multiple tools.
config	The *config* directory can be found in the following three locations: $HPEESOF_DIR/netlist_exp/config $HPEESOF_DIR/custom/netlist_exp/config $HOME/hpeesof/netlist_exp/config The *config* directory can contain tool–specific configuration files. These files will contain definitions that are used by the Front End Flow netlister to determine formatting (for example, `MAX_LINE_LENGTH = 1024`). The tool–specific configuration file naming convention is CNEX_config.<*tool*>. Other types of configuration files will normally need to be in the standard ADS configuration file locations (*$HPEESOF_DIR/config*, *$HPEESOF_DIR/custom/config*, *$HOME/hpeesof/config*, or the project directory). See Chapter 2, Configuration Files for more information about configuration files.
include	The files to automatically include in a netlist are specified within the tool–specific subdirectory of the *include* directory. See Chapter 4, Customizing a Netlister, for more information about automatically included files.

Adding Tools to Front End Flow

The Front End Flow netlister presents a drop-down list of available tools. The generated netlist is compatible with the selected tool. See Chapter 3 of the *Netlist Exporter* manual. Custom tools can be added to the list to meet specific netlist requirements.

The Need for Adding Tools

Front End Flow outputs netlists in an HSpice-like format, with the top level circuit represented as a subcircuit in the generated netlist. This format is ideal for a number of LVS tools, but is not well suited for simulators. The following are some reasons that an additional tool may be required:

- The design tool used does not support HSpice-like formats.
- Separate component definitions are required for components that do not have identical netlist representations for separate tools (even if both tools are able to utilize the default HSpice-like format).

Adding a Tool

Adding a new tool requires knowledge of how that drop down list is populated.

The configuration variable *CNEX_COMPONENT_PATH* defines the locations for the tool component definition files. See Chapter 2, Configuration Files for more information on *CNEX_COMPONENT_PATH*. Each tool that is configured needs to have at least one subdirectory, <*tool*>, under a Front End Flow *netlist_exp/components* directory that contains the component definition files.

If a subdirectory is found under an *netlist_exp/components* directory, it is assumed that subdirectory represents the name of a tool for Front End Flow. For example, there are three Front End Flow standard subdirectories, *assura*, *calibre*, and *dracula*, in the directory *$HPEESOF_DIR/netlist_exp/components*. These tool names are present in the tools drop–down list in the dialogs.

Note If a custom tool is selected from the tools drop–down list, component definitions will then only be read out of component directories that end in the leaf directory <*tool*>.

To add a new tool to the tools drop–down list, use the following procedure:

1. Make a new directory, <*tool*>, in a *components* directory. The name of the new directory will be the tool name displayed in the tools drop–down list.

   The <*tool*> directory needs to be located under one or more of the *netlist_exp/component* directories in *CNEX_COMPONENT_PATH*. It is not necessary to have <*tool*> located under every *netlist_exp/component* directory.

   The next time a Front End Flow dialog is called, it will have *tool* in the tools list.

2. If the default netlist exporting format is inappropriate for the new tool, it will be necessary to create custom AEL code that will support the new tool. Refer to Chapter 4, Customizing a Netlister for the process to write custom AE code. Place the custom AEL code in a directory called <*tool*> under *$HOME/hpeesof/netlist_exp/ael*. The Front End Flow netlister will look for files to support customization in that directory.

Chapter 2: Configuration Files

There are several text configuration files that define variables for the default Front End Flow netlister and variables for specific tools. This chapter covers the location of configuration files and which variables can be set for Front End Flow.

Configuration Files Used with Front End Flow

The following four configuration files contain pertinent Front End Flow information:

de_sim.cfg	The *de_sim.cfg* file is the PDE configuration file. This file is used to load Front End Flow.
CNEX.cfg	The *CNEX.cfg* file is the Front End Flow configuration file. This file contains information used by the core functions of the Front End Flow netlister.
<tool>.cfg	The *<tool>.cfg* configuration file contains specific data for a tool. The configuration data consists of options that are output into netlists for a particular tool.
CNEX_config.<tool>	The *CNEX_config.<tool>* configuration file contains variables that define certain global variables that are used to create netlists for a particular tool's format.

Configuration File Locations

The configuration files can be located at the following locations:

de_sim.cfg

This file can be located in one or more of the following locations (high to low priority):

- The project directory
- $HOME/hpeesof/config
- $HPEESOF_DIR/custom/config
- $HPEESOF_DIR/config

Front End Flow will follow the above priority, for example a variable in the project directory *de_sim.cfg* overwrites the value that is defined in *$HPEESOF_DIR/config/de_sim.cfg*.

CNEX.cfg

This file can be located in one or more of the following locations:

- The project directory
- $HOME/hpeesof/config
- $HPEESOF_DIR/custom/config
- $HPEESOF_DIR/config

Note Do not modify variable values within the project directory *CNEX.cfg* file. The project directory CNEX.cfg will be updated with values automatically. User modifications may be overridden.

<tool>.cfg

This file can be located in one or more of the following locations:

- Design kit directories
- $HOME/hpeesof/netlist_exp/config
- $HPEESOF_DIR/custom/netlist_exp/config
- $HPEESOF_DIR/netlist_exp/config

Note Do not modify variable values within the project directory *CNEX.cfg* file. The project directory CNEX.cfg will be updated with values automatically. User modifications may be overwritten.

CNEX_config.<tool>

This file can be located under any of the *netlist_exp* directories defined in the *CNEX.cfg* directory. Therefore, files can be in *{%CNEX_INSTALL_DIR}/config*, *{%CNEX_CUSTOM_DIR}/config*, or *{%CNEX_HOME_DIR}/config*. In addition, design kit directories are checked for a *netlist_exp/config* directory; therefore, you can place a *CNEX_config.<tool>* configuration file in a design kit.

Configuration File Descriptions

The configuration files are defined as follows:

de_sim.cfg

Modify *only* the *USER_MENU_FUNCTION_LIST* configuration variable within *desim.cfg*. This variable defines the custom menus within the ADS product. To add the Front End Flow menu, the following line should be added to *de_sim.cfg* if it does not already exist:

```
USER_MENU_FUNCTION_LIST=app_add_user_menus;app_add_cnex_menus
```

The function *app_add_user_menus* is a default function which you can be create. Call this function first in the list to maintain the default ADS operability. The function *app_add_cnex_menus* will add the ADS Front End menu to the schematic tools menu.

If the value *USER_MENU_FUNCTION_LIST* is already set, add *app_add_cnex_menus* to the existing list (if not already present). The list is delimited by semicolons.

Note If you add *app_add_cnex_menus* to the *USER_MENU_FUNCTION_LIST*, and you do not get the Front End Flow menu, make sure that *USER_MENU_FUNCTION_LIST* is not defined in a higher priority *de_sim.cfg file* without the *app_add_cnex_menus* function call.

Priority Override Example

If you have *USER_MENU_FUNCTION_LIST=app_add_user_menus* in the file *$HOME/hpeesof/config/de_sim.cfg*, and *USER_MENU_FUNCTION_LIST=app_add_user_menus;app_add_cnex_menus* in the file *$HPEESOF_DIR/custom/config/de_sim.cfg*, you will not see the menu. The value in the home directory *de_sim.cfg* file will take priority over the value set in the custom directory *de_sim.cfg* file.

CNEX.cfg file

The *CNEX.cfg* file contains all of the configuration variables for starting Front End Flow, and for specifying paths where component definitions and AEL can be found for different tools. The following variables can be set in *CNEX.cfg*:

CNEX_TOOL

The *CNEX_TOOL* configuration variable specifies which netlist format will be created. It is normally set by the Front End Flow dialogs, and stored within the *CNEX.cfg* file that is generated in the current project directory. Setting it in the default configuration file in *$HPEESOF_DIR/config* will set a default value for the initial tool to use with Front End Flow.

CNEX_CUSTOM_DIR, CNEX_HOME_DIR, CNEX_INSTALL_DIR

ADS files are ordinarily stored in the following locations:

- $HPEESOF_DIR

  The *$HPEESOF_DIR* directory contains the ADS program as it was installed from the CD packages.

  Note Do not modify the contents of *$HPEESOF_DIR*. Patches will always install to *$HPEESOF_DIR*, overwriting any customizations.

  The *CNEX_INSTALL_DIR* configuration variable specifies the location of the installation files. The Front End Flow installer will always install the Front End Flow code to *$HPEESOF_DIR/netlist_exp*. To move the files from that location to a directory that is not in the ADS main directory tree, modify *CNEX_INSTALL_DIR* accordingly. This may be necessary if you wish to maintain multiple versions of Front End Flow simultaneously without using multiple ADS installations.

- Custom directory under $HPEESOF_DIR

  The custom directory facilitates site wide customizations and settings. The custom directory contents are not overwritten when code patches are installed.

  The *CNEX_CUSTOM_DIR* configuration variable specifies the location of the custom directory storage for the Front End Flow product. It can be set to point at any directory location. The value defaults to *$HPEESOF_DIR/custom/netlist_exp*.

- $HOME/hpeesof

  The hpeesof directory in the user's home directory facilitates user specific customizations and settings. The custom directory contents are not overwritten when code patches are installed.

The *CNEX_HOME_DIR* configuration variable specifies the location of the home directory storage for the Front End Flow product. It can be set to point at any directory location. The value defaults to *$HOME/hpeesof/netlist_exp*.

CNEX_EXPORT_FILES

The *CNEX_EXPORT_FILES* configuration variable specifies the location of the AEL files *nexGlobals* and *cnexNetlistFunctions*. The default value, *{%CNEX_INSTALL_DIR}/ae*l, is the installation directory defined by the *CNEX_INSTALL_DIR* variable.

Note Do not modify the default value of *CNEX_EXPORT_FILES*.

CNEX_STARTUP_AEL

The *CNEX_STARTUP_AEL* configuration variable specifies the AEL file name to load during ADS boot-up. The default value is *{%CNEX_EXPORT_FILES}/cnexexport*.

Note Do not modify the default value of *CNEX_STARTUP_AEL*. If the variable does not point to a valid file named *cnex_export*, Front End Flow will not be loaded.

CNEX_DESIGN_KIT_PATH

The *CNEX_DESIGN_KIT_PATH* variable will be updated within the current project directory's *CNEX.cfg* file when design kit software is installed. The path variable will define only the paths to kits that contain a *netlist_exp* directory with a *components* subdirectory for the currently active tool.

Note The value of this variable within the current project directory's *CNEX.cfg* file will overwrite the variable value in any other *CNEX.cfg* files.

CNEX_DESIGN_KIT_AEL_PATH

The *CNEX_DESIGN_KIT_AEL_PATH* configuration variable will be updated within the current project directory's *CNEX.cfg* file when design kit software is installed. The path variable will define only the paths to kits that contain a *netlist_exp* directory with an *ael* subdirectory for the currently active tool.

Note The value of this variable within the current project directory's *CNEX.cfg* file will overwrite the variable value in any other *CNEX.cfg* files.

CNEX_EXPORT_FILE_PATH

The *CNEX_EXPORT_FILE_PATH* configuration variable specifies the locations that will be searched for Front End Flow AEL files. During netlist exporting, ADS will search the path and load files with the names *cnexGlobals* and *cnexNetlistFunctions*.

AEL has a single name space for all of its variables and functions. When a duplicate function or global variable is found in a file, it will overwrite the value that is currently in memory. This allows customizations to be done by creating new functions or global variables in a *cnexGlobals* file or *cnexNetlistFunctions* file. It is not necessary to duplicate all of the code in the earlier files, only the code that requires modification. It is also possible to add new functions or variables in the leaf files. This code will go into the single name space for AEL, and can be accessed globally in the same manner that the core Front End Flow API functions can be accessed.

Note The path order determines the priority. Place the paths with the *lowest* priority first in the list. Files that are located later in the path will then be able to overwrite the settings that exist in the earlier files.

The default value of *CNEX_EXPORT_FILE_PATH* includes the definition for *CNEX_DESIGN_KIT_AEL_PATH*. It is not necessary to update this value to hard code the design kit locations.

This variable will also be used to load the GUI options file, *cnexOptions*.

CNEX_COMPONENT_PATH

The *CNEX_COMPONENT_PATH* configuration variable specifies the locations that will be searched for Front End Flow component definition files. When a component definition file is encountered during netlist exporting, a name will be constructed consisting of the component design name, with a suffix of *.cnex*.

The component path will be searched, until the first instance of a file with the name *<component>.cnex* is found. That component definition file will then be read and used to format the instance for the netlist.

Note The path order determines the priority. Place the paths with the *highest* priority first in the list.

The default value of *CNEX_COMPONENT_PATH* includes the definition for *CNEX_DESIGN_KIT_AEL_PATH*. It is not necessary to update this value to hard code the design kit locations.

Tool Configuration Files

Every tool that is used with Front End Flow can potentially have its own netlist exporting options and netlist exporting options dialog. Because the options dialogs will be custom written, there is no set format for these configuration files. The following are some recommendations for the files:

- Make the option configuration variable match the option name.
- Booleans should be output as 0 and 1.
- Lists must be converted into strings with a delimiter.

Note Do not output the values using the *identify_value* function This will make it difficult to interpret lists in the configuration file.

CNEX_config Configuration File

Every tool supported by Front End Flow should have an *CNEX_config.<tool>* file created for it. This file should be placed in the *config* directory of

%CNEX_INSTALL_DIR. The settings in the *CNEX_config* file specify certain global variables that are used by the Front End Flow netlister. The following are the valid *CNEX_config* variables:

CASE_INSENSITIVE_OUTPUT = TRUE | FALSE

The ADS schematic environment is case sensitive. Most Spice formats are not case sensitive. If *CASE_INSENSITIVE_OUTPUT* is set to *TRUE*, the netlister will map all netlist node names and instance names to lower case. The netlister will then check for conflicting names when the final netlist is created. If conflicts are found, a warning will be displayed that indicates the conflicting names.

The *CASE_INSENSITIVE_OUTPUT* default value is *TRUE*.

Note Additional name mapping will not be performed to resolve name conflicts. Case sensitivity issues (name conflicts) will require manual name edits within schematics.

COMPONENT_INSTANCE_SEPARATOR

The *COMPONENT_INSTANCE_SEPARATOR* configuration variable specifies a separation charter to be inserted in between the Spice component type character and the ADS instance name. This can be used to increase readability of the final netlist (some Spice dialects use a leading character to designate the component type. For example *R* designates a resistor).

For example, specifying the underscore character, _ , would generate a Spice netlist instance name of *R_R1* for an ADS resister component with the name *R1*.

The *COMPONENT_INSTANCE_SEPARATOR* default value is null.

EQUIV = <node1> <node 2>

The *EQUIV* configuration function combines two nodes, <*node1*> and <*node2*> together into <*node1*>. This function is useful for nodes that are connected (common) on the schematic.

As many *EQUIV* lines can be placed in the configuration file as are necessary to define all of the equivalent node names. During netlist exporting, any time <*node 2*> is encountered, it will be renamed to <*node 1*>. In addition, this list is built internally by the ignore instance netlist exporting functions.

EXPRESSION_MAPPING = <ADS name> <netlist name>

The *EXPRESSION_MAPPING* configuration function maps ADS expressions, <*ADS name*>, to a user defined expression name, <*netlist name*>.

When a parameter value is encountered, it will be searched for expressions. Any expression that is found in the expression mapping list will be converted from the ADS expression name, <*ADS name*>, to the target netlist expression name, <*netlist name*>.

For example, in HSpice, the natural logarithm function is *log*, in ADS it is *ln*. To have the netlister change all instances of *ln* to *log*, add an expression mapping line of *EXPRESSION_MAPPING = ln log* in the *CNEX_Config.hspice* configuration file.

Place one *EXPRESSION_MAPPING* line for each expression to be mapped within the configuration file.

EXPRESSION_START, EXPRESSION_END

The *EXPRESSION_START* configuration variable specifies a expression start character and the *EXPRESSION_END* configuration variable specifies the expression end charter to be used for HSpice and PSpice netlist generation. (HSpice and PSpice require special characters to designate the start and end of an expression.)

ADS allows expressions in its parameter values, it may be necessary to have those expressions prefixed with the expression start designator, and suffixed with the expression end designator.

For example, if *EXPRESSION_START* is set to ‘ and *EXPRESSION_END* is also set to ‘, and an instance value of *R=RVal1+RVal2* is specified on an ADS resistor, the value output to the netlist would be *R='RVal1+RVal2'*.

The default is to have no expression start or end designators.

GROUND

The *GROUND* configuration variable specifies the global ground node name.

In ADS, node *0* is the global ground node. All instances of node *0* are mapped to the value of the value of *GROUND*.

For example, if it is known that the layout uses *GND* as the ground node, set the *GROUND* value to *GND*. All nodes named *0* will be output as *GND*.

The default is no node *0* name mapping.

LINE_COMMENT

The *LINE_COMMENT* configuration variable specifies the character to output at the beginning of comment lines. The default value is *.

LINE_CONTINUATION_CHARACTER

The *LINE_CONTINUATION_CHARACTER* configuration variable specifies the character used to declare a line continuation.

If the maximum line length for the netlist is exceeded, a line continuation will be output. Different tools support different methods for declaring a line continuation. This will either be output at the end of the current line, or at the beginning of the next line, depending on the *LINE_CONTINUATION_MODE* variable.

The default *LINE_CONTINUATION_CHARACTER* value is +.

LINE_CONTINUATION_MODE

The *LINE_CONTINUTATION_MODE* configuration variable specifies how the continuation character will be output when a continuation line is required. A value of *0* will be output the continuation character at beginning of the next line. A value of *1* will be output the continuation character at the end of the current line. Values above 1 are reserved for future use.

The default *LINE_CONTINUTATION_MODE* value is *0*.

MAX_LINE_LENGTH

The *MAX_LINE _LENGTH* configuration variable specifies the maximum line length that will be output before a line continuation character is output.

The default *MAX_LINE _LENGTH* value is 1024 characters.

NUMERIC_NODE_PREFIX

The *NUMERIC_NODE_PREFIX* function adds a specified prefix to system defined node names.

ADS supports the following two type of node names:

- Wire label

The node names are explicitly defined by the user. The *NUMERIC_NODE_PREFIX* function ignores these node names.

- System node name

  Node names are system generated for any net that does not have an explicit label, or is not attached to ground. The system node names are numbers only. If you are using a tool that does not support numeric node names, use the *NUMERIC_NODE_PREFIX* function to add a prefix to all system defined node names.

The default value for the node prefix is *_net*.

Note In ADS netlists, the *_net* prefix designates that the node name will not be saved to a dataset.

For example, if the *NUMERIC_NODE_PREFIX* is set to *_net*, and a node is encountered in ADS with the system defined node name *27*, the netlister will output the value *_net27* for the new node name.

SCALAR_TO_SCIENTIFIC = FALSE | TRUE

The *SCALAR_TO_SCIENTIC* function maps scalar quantities into scientific notation.

The *SCALAR_TO_SCIENTIC* function is useful if your tool's netlist format does not support scalars. See *SCALAR_UNIT_MAPPING* for information on customizing scaler mapping.

The *SCALAR_TO_SCIENTIC* default value is *FALSE*. (No function line present equals *FALSE*.)

For example, if *SCALAR_TO_SCIENTIC* is set to *TRUE*, *1n* would be output as *1e-9*.

SCALAR_UNIT_MAPPING = <ADS Scalar> <netlist scalar>

The *SCALAR_UNIT_MAPPING* function maps specified scalar quantities, *<ADS Scalar>*, into the specified representation, *<netlist scalar>*.

Use the following mapping guidelines:

- Place one line into the file for each scalar that is to be mapped.

- Include units and scaling value (for example, *MHz)* for the ADS scalar quantity, <*ADS Scalar*>.
- If you want nothing output for the scalar, leave the second field blank (for example, *SCALAR_UNIT_MAPPING = A*).

When the value is output to the netlist, any occurrences of the ADS scalar/unit will be replaced with the netlist equivalent.

Chapter 3: Component Definitions

In ADS, a component is a symbol that has a specific set of parameters and terminals. It may also have a related schematic or layout.

Every component makes a single call to the *create_item()* function which defines the name of the symbol file, the schematic file, and the parameters. The *create_item* call causes ADS to create a uniquely named component. These uniquely named components can go into ADS schematic hierarchies that are netlisted and simulated in the ADS simulator.

For netlist exporting, the terminals are determined by accessing the schematic database file and the symbol database file. These do not have to be the same file in ADS.

Component Definition Files

Component definition files are ASCII text files in the CNEX_COMPONENT_PATH that contain variables that determine how to netlist a component for a particle tool. Variable names are not case sensitive, but their values are. The variables can be in any order in the file.

Note The file <*ADS component name*>.*cnex* must be in the appropriate tool directory for the component definition file to work.

ADS subcircuits do not need to have a component definition file because they inherit the default subcircuit format. However, if an ADS component is not a hierarchical design, it must have a definition file.

Component Definition File Variables

Netlist_Function

This variable contains the name of the instance function to be called to format an instance of a component for a netlist.

You have three options for choosing an instance function:

- You can use the functions that come with Front End Flow (They are in the "Instance Netlist Exporting Functions" on page A-1).
- You can enter the name of a custom written function.
- You can leave the *Netlist_Function* out of the component definition file. In this case, Front End Flow automatically uses the function *cnexSubcircuitInstance* if the component is a subcircuit, and *cnexUnknownInstance* if it is a primitive.

Syntax

```
Netlist_Function = <function>
```

Example

```
Netlist_Function = cnexNetlistInstance
```

Component_Name

This variable specifies the name for the component instance in the netlist.

In general you can assign any name, but there are some rules for specific cases:

- If you use the netlist function *cnexSubcircuitInstance*, you can type in `subcircuit` for the component name. Or, you can assign any other name.
- If you want to use the value of a parameter as the component name, use `@` followed by the name of the parameter.
- If you use Spice, you can use a single letter for the component name.

Syntax

```
Component_Name = <name>
```

Example

```
Component_Name = R
```

Note You must always assign a value to this variable in the Component Definition File.

Terminal_Order

This variable specifies the order for outputting component pins for netlist exporting.

- You can specify the order in either pin numbers or pin names, but you cannot use both together.
- If you do not assign an output order, Front End Flow uses the ADS pin output order which is sequential by pin number.

Note Other tool vendors may use a pin output order that is different from the ADS order. Check the documentation for information on the correct order when you specify this variable.

Note Many of the standard ADS components do not have pin names, so you must assign pin number output order.

Syntax

```
Terminal_Order = <value>
```

Example

```
Terminal_Order = 1 2
```

Parameters

This variable specifies the parameters to output to the netlist for an instance. Front End Flow outputs the parameters in the order you list them.

- You should specify the parameters as a space delimited list.
- If you do not specify this variable, no parameters will be output for the instance.

Syntax

```
Parameters = <parameter> <parameter>
```

Example

```
Parameters = R _M Model Width Length
```

Parameter_Name_Mapping

This variable maps an ADS parameter name to a netlist name.

If you do not want to output the ADS parameter name to, leave *Netlist Name* blank.

If you want the ADS parameter name to be the same as the netlist name, do not assign this variable.

Syntax

```
Parameter_Name_Mapping = <ADS Name> (<Netlist Name> |)
```

Example

R1 is an instance in ADS. It connects to *nodes _net1* and *ground*, and has the parameters R=50 and _M=2:

```
Parameter_Name_Mapping = R
```

- The ADS parameter name *R* is not mapped to a netlist name.

```
Parameter_Name_Mapping = _M m
```

- The ADS parameter name *_M* is mapped to the netlist name *m*.

Resulting netlist output for Dracula is the following:

```
RR1 _net1 0 50 m=2
```

- *R* is not mapped, therefore it is output as 50 in the instance line.
- *_M* is mapped to *m*, therefore it is ouptut as *m=2* in the instance line.

Parameter_Type_Mapping

This variable specifies the AEL mapping function for an ADS parameter.

- If you specify an AEL mapping function, the ADS parameter value will be passed to that function.The function returns the value that needs to be output to the net list.
- If you do not specify an AEL mapping function for a parameter, no mapping will be done.

Syntax

```
Parameter_Type_Mapping = <ADS parameter> <AEL mapping function>
```

Example

A custom resistor component named *myAdsRes* is in ADS. It has two models, *myAdsRes1* and *myAdsRes2*. The Dracula LVS rule file extracts these models as *R1* and *R2*. For correct netlist output, you must map *myAdsRes1* to *R1* and *myAdsRes2* to *R2*. In *myAdsRep.cnex*, the component definition file, add the following line:

```
Parameter_Type_Mapping = Model myAdsResMap
```

When the circuit is netlisted, the function *myAdsResMap* will be called when the Model parameter is output. The function will return the appropriate values for output for Dracula.

Note There are no standard AEL type mapping functions. You must write them. See "Adding Value Mapping Functions" on page 4-2 for information. Check the documentation for the tool you are using to verify the value outputs that it needs.

Component Definition File Editing

There are the following two ways to edit component definition files:

- You can use a text editor to directly edit the component definition files.
- You can use the Component Definition Editor, see "Component Definition File Setup with the GUI" on page 3-5.

Component Definition File Setup with the GUI

The *Component Definition Editor* (Figure 3-1) graphical user interface is available from the **Tools > Netlist Export** menu. The Editor enables you to specify all the fields needed for a component definition.

The *Component Definition Editor* can also read in ADS item definitions so that it can automatically populate the fields for components that do not already have a definition. For defined components, it can provide information about available parameters in the ADS item definition. The *Component Definition Editor* automatically updates the component definition table, preventing the use of old definitions.

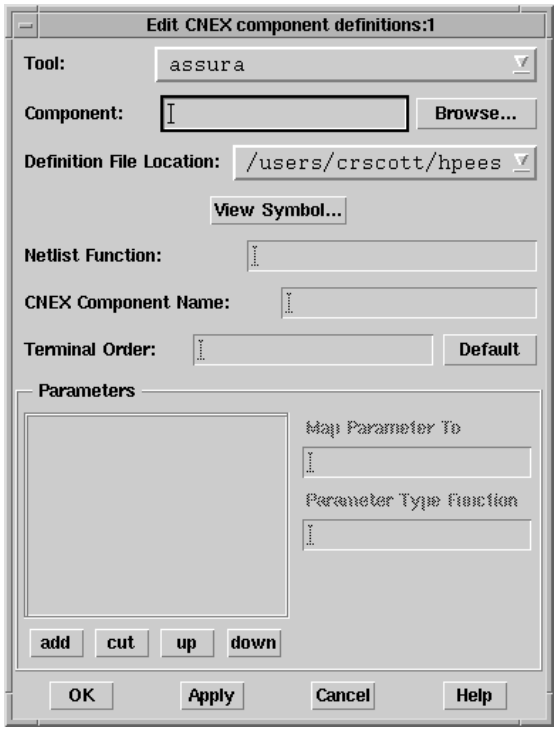


Figure 3-1. Component Definition Editor

Component Definition Editor Items

Tool:	Choose the tool you want from drop-down menu. The default tool when you start the *Component Definition Editor* is the last one specified in the component edit or the netlist dialog.
Component:	Type in the component name.
Browse...	If you do not know the name of the component you want to edit, press Browse... to bring up the Library Browser Winow. The Editor automatically reads in the definition of the component you select.
Definition File Location	This drop-down menu contains the path for the component definition file. You may edit the path if you do not find the one you want in the menu. When you edit the path, make sure you specify a file in a site wide location and not a user location.

Component Definition Editor Items

Definition File Location: Continued	If there is no file at the specified location, the *Component Dialog Editor* will follow the component definition path contained in the CNEX_COMPONENT_PATH configuration variable and use the first definition file it finds. If there is no definition file, Front End Flow will create one based on the information in the AEL *create_item* call.
The following fields are for entering variable vales in the component definition file.	
Netlist Function:	Choose the name of the function you want to use to format an instance of the component for netlist exporting. "Instance Netlist Exporting Functions" on page A-1 contains the list of the default functions that come with Front End Flow. You can also write your own netlist export function. If you do not choose a function, Front End Flow will use the *cnexSubcircuitInstance* if the component is a subcircuit, or *cnexUnknownInstance* if the component is a primitive.
CNEX Component Name:	Type in the name you want for the component instance. If use the *cnexSubcircuit* netlist function (see **Netlist Function:** above), you can use *subcircuit* for a name. If you are using Spice, you can use a name that is just one character. In this case, the tool for which you are exporting the netlist will determine the name.
Terminal Order:	This field specifies the pin output order for netlist exporting. The ADS pin order for a component, which is sequential by pin number, is the default. You should check with the documentation for other vendor tools because the pin order for those tools may not be the same as the ADS default. You can use either pin numbers or pin names to set the terminal order. You cannot mix the two. You must use pin numbers with ADS components because many of them do not have pin names.

Component Definition Editor Items

Parameters:	This list box contains the parameters you want to output. The parameters are output in the order listed. To move a parameter forward in the output list, select the parameter and click the **up** button. To move a parameter back in the list, select the parameter and click the **down** button. To remove a parameter from the list, select the parameter and click the **cut** button. To add a parameter, click the **add** button. The *Add a parameter for netlist exporting* dialog box will appear. This dialog box contains the list of parameters for the current ADS item definition that are not already in the parameters list. If you do not find the parameter you need in the list, you must add the parameter to the ADS item definition for the component. If the *Component Definition Editor* finds no definition for the current component, then the Editor will put all parameters that are netlisted for ADS in the *Parameters* list box. Each parameter has two properties used for netlist exporting: a name mapping and a value mapping. Click on a parameter in the list to see its properties and edit them as appropriate.
Map Parameter To	This field displays the netlist name for the parameter selected in the *Parameters* list box. If you do not want any mapping for a parameter, the name in this field should be the same as the parameter name. If you want to change an existing netlist name in this field, edit the name in the field. If you do not want a netlist name for a parameter, make this field blank.
Parameter Type Function	This field displays the AEL mapping function for the parameter selected in the *Parameters* list box.

Component Definition Editor Items

Parameter Type Function Continued	If you specify an AEL mapping function in this field, the ADS parameter value will be passed to that function.The function returns the value that needs to be output to the net list. If you leave this field blank, no mapping will be occur. Front End Flow has no standard type mapping functions. You must write your own AEL mapping functions. See “Adding Value Mapping Functions” on page 4-2 for information

Component Definition Editor Procedure

1. Select the tool from the *Tool:* drop-down menu.
2. Type the component name in *Component:*, or select it by clicking **Browse**.
3. Press **Tab** or select the next field in the *Editor* and Front End Flow will read the component definition into the *Editor*.
4. Use **Definition File Location:** to specify the path to the component definition file.
5. Specify the component definition file variable values in the remaining fields of the GUI.

Chapter 4: Customizing a Netlister

You can customize the Front End Flow netlister. You can automatically include files and add value mapping functions to those described in Appendix A, Front End Flow Functions. You can also add netlist exporting functions to those already in the Front End Flow API. And, you can override many of the functions listed in Appendix A, Front End Flow Functions.

Setting Up Automatically Included Files

The simplest method of customizing a netlister is setting up files that are automatically included in the final netlist.

Note Refer to the *Netlist Exporter* manual for the process for excluding included files.

The Include File Path

You should include files that set the options for a particular process or create the subcircuits necessary for a foundry process.

Any file in the following directories will be included in the Front End Flow netlist, unless you set them up to be excluded:

- {%CNEX_INSTALL_DIR}/include/{%CNEX_TOOL}
- {%CNEX_CUSTOM_DIR}/include/{%CNEX_TOOL}
- {%CNEX_HOME_DIR}/include/{%CNEX_TOOL}
- Any design kit path that contains the directory netlist_exp/include/{%CNEX_TOOL}

Example 1: Including a File Site Wide

You are using the Dracula tool and have set the variable *CNEX_INSTALL_DIR* to be *$HPEESOF_DIR/netlist_exp*. Place the file *standard.inc* in the directory *CNEX_INSTALL_DIR/include/dracula*. Whenever you create a netlist for Dracula, *standard.inc* will be included. If you use a different tool, the file will not be included.

CNEX_INSTALL_DIR is available to all site users, therefore *standard.inc* will be an included file for all who use the ADS installation on that site. If the ADS installation is on a shared drive that all users access, the included file will be automatically available to those users.

Example 2: Foundry Kit Include File

You have a foundry design kit from Foundry A that has the file *foundryAOptions.inc*, and you are using Dracula. Place the file in *netlist_exp/include/dracula*. When you run Dracula, *foundryAOptions.inc* will be automatically included.

Adding Value Mapping Functions

When the tool you have chosen uses a different type of name than does ADS, or when that tool uses different parameters or values than does ADS, you must write an AEL value mapping function to supply the tool with the correct output.

Case 1: Name Mapping

Use ADS model names that indicate what the component is, such as *myAdsRes1*. However, Dracula only allows two character model names. Map the ADS name to a two character name that Dracula recognizes.

Case 2:Parameter Mapping

A single parameter in ADS may need to map to multiple parameters in another tool, or multiple parameters in ADS may need to map to a single parameter.For example, in ADS the *V_1Tone* device has the parameters voltage and frequency. If you use the HSpice tool, you must map those parameters to the single SIN function for the correct HSpice output.

Case 3: Parameter Value Mapping

You may need a function that performs an operation on a parameter that is a value in ADS and returns a value that is mapped to a parameter in another tool. For example, you want to map the ADS temperature parameter to the differential *dtemp* parameter in HSpice. This requires a function that sets the ADS parameter value to the absolute temperature set in ADS and subtracts the circuit temperature. The function returns a value that contains the differential temperature and maps it to *dtemp*.

Function Prototype and Example

You must write all AEL value mapping functions. All value mapping functions receive an ADS parameter value and return the correct tool value. They all use the following prototype:

```
defun <function name> (value)
{
<your code here>
return(<new value>);
}
```

In addition to the parameter value you pass to the function, you must set the following three global variables:

- *cnexCurrentRep*

  This is a handle to the schematic that is currently being processed. The handle can be used to obtain data about other instances in the circuit.

- *cnexCurrentInst*

  This is a handle to the instance that is currently being processed. The handle can be used to get instance data, or data for other parameters.

- *cnexCurrentParam*

  This is a handle to the parameter that is currently being processed. Parameter attributes can be obtained by referencing this handle.

Example1: Writing a Type Mapping Function

```
defun myFoundryAddQuotes (value)
{
decl newValue=value;

if(is_string(value))
{
newValue=strcat(“\””, value, “\””);
}

return(newValue);
}
```

This function adds quote marks around the ADS value passed to it.

Adding the New Netlist Function

To add an instance netlist function, edit the component definition file for your component by adding a line with the syntax: *Parameter_Type_Mapping = <param> <function>*.

Example1: Adding the Function to the Component Definition File

You have an ADS component named *myNpn* which has a parameter named *Model*. You are using the type mapping function *myFoundryAddQuotes*.

1. Open the file the component definition file *myNpn.cnex*.
2. Add the line: *Parameter_Type_Mapping = Model myFoundryAddQuotes*

When an instance of *myNpn* component is netlisted, the parameter value for *Model* will have double quotes around it.

Placing the Type Mapping Function

The configuration variable CNEX_EXPORT_FILE_PATH specifies the path where ADS searches for AEL files. During netlist exporting, ADS loads files named *cnexGlobals* and *cnexNetlistFunctions* located in that path.

AEL has a single name space for all of its variables and functions. All files named *cnexGlobals* or *cnexNetlistFunctions* in any path contained in the CNEX_EXPORT_FILE_PATH use this space. Therefore any AEL function has access to all AEL variables and functions.

These functions can be added in at any time. Every time the Front End Flow netlister is executed, all of the *cnexNetlistFunctions* and *cnexGlobals* AEL files are loaded.

Note See Chapter 2, Configuration Files for more information on CNEX_EXPORT_FILE_PATH.

Example 1: Placing the Function

You want to place the function *myFoundryAddQuotes* so that Front End Flow can use it. Add a file named *cnexNetlistFunctions* to the directory *{%CNEX_INSTALL_DIR}/ael/{%CNEX_TOOL}*. If you wrote the function for HSpice, the directory would be *$HPEESOF_DIR/netlist_exp/ael/HSpice*. The next

time that a Front End Flow netlist is generated, the *myFoundryAddQuotes* function will be available.

Validating a Type Mapping Function

Once the files have been loaded by creating a Front End Flow netlist, you can validate your function interactively using the ADS *Command Line* window. Bring up by selecting the menu option **Options > Command Line** from the main window. Enter an AEL function in the command line and you will see the value the function returns.

Validating Functions that Do Not Use Global Variables

Simple type mapping functions that do not require global variables to be set can be tested rapidly by using the *info* command and typing the function into the *Command Line*. This is shown in Figure 4-1.

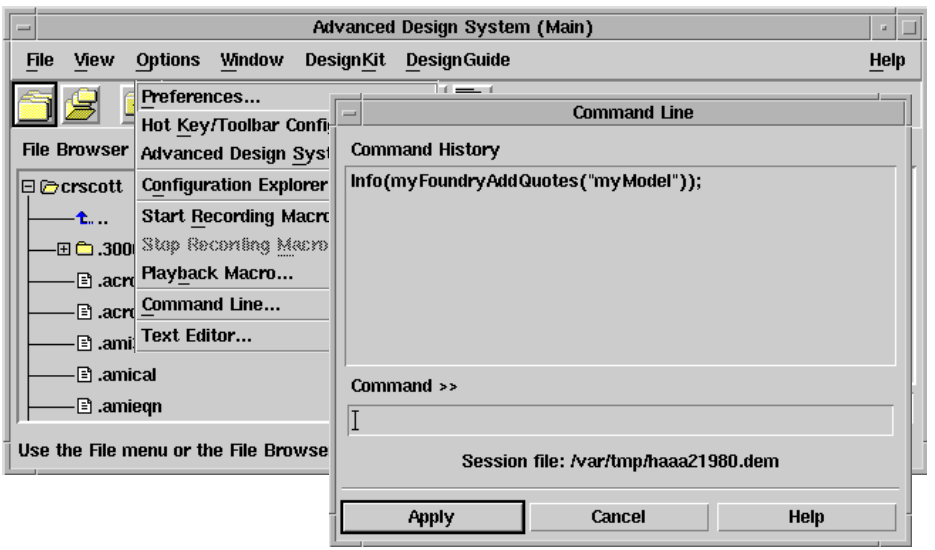


Figure 4-1. Accessing the AEL Command Line to Verify Functions

1. Enter the function in the *Command Line* window.
2. The resulting return value appears in the *Information* window when you use the *info dialog*.

Note The *info dialog* is modal and stops the execution of any further AEL operations.

Validating Functions That Do Use Global Variables

Use the *fputs* function and specifying *stderr* as the destination of the output. The *fputs* command using *stderr* will output text to the *xterminal* window from which ADS started.

1. Type the command **fputs(stderr, myFoundryAddQuotes("myModel"))** in the command line.
2. The result will display in the *xterminal* window.

Note If you are running Front End Flow on a PC, be aware that it has no *xterminal* window. However, if you run ADS with the -d option you will have a *debug* window in which you can see the *stderr* output. The *debug* window will also display inter-process function call text.

If the function needs global variables, add `fputs(stderr, value)` calls in your code.

Using Some Other Debugging Tips

- The *identify_value* function converts any AEL expression passed to it into a string. It is useful if you have values that are set to NULL, or if you are debugging lists because the *fputs* command only outputs strings.
- For formatted debugging output, you use `fprintf(stderr, ...)`. The *fprintf* function in AEL utilizes the same formatting strings as the Ansi C *fprintf* function.

Adding New Netlist Exporting Functions

The Front End Flow API provides 8 instance netlist exporting functions. Although these functions provide the correct output for nearly any ADS instance, there are some situations where you must write your own function in order to have the correct output. Some examples of this are outputting multiple instead of single components to a netlist file, and outputting a library or directive with a component.

Function Prototype and Example

All instance netlist exporting functions must have the following function prototype:

```
defun <function name> (instH, instRecord)
{
<your code here>
return(<string>);
}
```

- The parameter *instH* is the handle to the instance that is currently being formatted.
- The *instRecord* parameter is a list of lists that contains the information obtained by reading the component definition file for the instance.
- The *return* string is the value you wish to be output to the netlist file.

Example1: Writing a Type Mapping Function

```
defun myNpnInstance (instH, instRecord)
{
decl net=“”;
decl netReturn=cnexNetlistInstance(instH, instRecord);

net=strcat(“.lib ‘MYMODELS’ npn\n”, netReturn);

return(net);
}
```

The function *myNpnInstance* causes the *myNpn* component to output a *.lib* statement when the instance is netlisted. The *myNpnInstance* function calls the *cnexNetlistInstance* function to get the instance netlist exporting line. The result from *cnexNetlistInstance* is then concatenated with the *.lib* statement. The result is then returned.

Using the New Netlist Function

To use an instance netlist function, you must edit the component definition file for your component by adding a line with the syntax: *Netlist_Function = <function>*.

Adding the Function to the Component File

Open *myNpn.cnex*, component definition file for *myNew*, and change the line *Netlist_Function* is changed so it reads:

Netlist_Function = myNpnInstance

The *myNpn* component will now use the newly created instance netlist exporting function.

More detailed examples can be found in Chapter 6, Hspice Netlister Example.

Placing a New Netlist Exporting Function

The configuration variable CNEX_EXPORT_FILE_PATH specifies the location where ADS searches for AEL files. During netlist exporting, ADS follows the path and loads files named *cnexGlobals* and *cnexNetlistFunctions*.

AEL has a single name space for all of its variables and functions. All files named *cnexGlobals* or *cnexNetlistFunctions* in any path contained in the CNEX_EXPORT_FILE_PATH uses this space. Therefore any AEL function has access to all AEL variables and functions.

These functions can be added in at any time. Every time the Front End Flow netlister is executed, all of the *cnexNetlistFunctions* and *cnexGlobals* AEL files are loaded.

Note See Chapter 2, Configuration Files for more information on the CNEX_EXPORT_FILE_PATH.

Adding the New Instance Netlist Exporting Function to a File

You want to place the function *myNpnInstance*. Add a file named *cnexNetlistFunctions* to the directory *{%CNEX_INSTALL_DIR}/ael/{%CNEX_TOOL}*. If you wrote the function for HSpice, the directory would be *$HPEESOF_DIR/netlist_exp/ael/HSpice*. The next time that a Front End Flow netlist is generated, the *myNpnInstance* function will be available.

Overriding Existing Front End Flow API Functions

Front End Flow provides API netlister functions for three tools: Dracula, Calibre, and Assura. Appendix A, Front End Flow Functions describes these functions. If you use a different tool, you should override the default API functions relevant to the correct

output for your tool. To override means to keep the name of the default API function, but to modify its code so that it provides the correct output for your tool.

It is better to override the existing API functions than to write a new one with a new name. The reason is that end users may need to modify standard ADS component definitions so that they can map into your foundry process. These standard ADS components use the default API function names. If you use new API function names, the end user will not know the use of the new function, unless you supply that information. If you override the default functions, the end user can use the standard Front End Flow documentation.

You can override the functions at any time. Every time the Front End Flow netlister is executed, all of the *cnexNetlistFunctions* and *cnexGlobals* AEL files are loaded.

Function Prototype

To override a Front End Flow API function, you must make sure to follow the function prototypes that are defined in Appendix A, Front End Flow Functions.

- Whenever you override, you must use the same argument list and return name as the default function.
- There are some core functions in Front End Flow that will call the API functions directly. Those functions cannot be overridden.

Note If the functions called by the Front End Flow core return incorrect names, or receive the wrong arguments, the Front End Flow netlister will error out and no netlist will be produced.

Example 1: Overriding the Top Cell Header Function

The function that controls how the top cell is output is the Front End Flow API function cnexOutputTopcellHeader. The values it receives are the design name and the design handle. For the Dracula-spice netlister, the function simply calls the subcircuit header function, which will cause a *.subckt* line to be created for the top cell as follows:

```
defun cnexOutputTopcellHeader(designName, dsnH)
{
    return(cnexOutputSubcircuitHeader(designName, dsnH));
}
```

For HSpice, the *.subckt* line is not required. In this case, override the default API function to output nothing for the top cell header:

```
defun cnexOutputTopcellHeader(designName, dsnH)
{
    return(“”);
}
```

Place this function into a file called *cnexNetlistFunctions*. Put this file in an HSpice subdirectory in one of the CNEX_EXPORT_FILE_PATH directories. When a netlist is created for HSpice, it will no longer use the default top cell function, it will use the API function that was modified for HSpice.

For more detailed examples of overriding API functions, refer to Chapter 6, Hspice Netlister Example.

Subclassing a Function Definition

You may want to simply add a feature to an existing API function instead of overriding it. This is called *subclassing* a function.

In AEL, everything can be accessed as a variable, including functions. If you create a variable and set the variable equal to a function, you can call the variable value just like it was the function name. To save the function so you can access it later, you need to define the AEL variable as a global variable.

You can also create a variable and store a function to the variable, then override the function. The old definition is still in memory, and can be accessed through the variable. To save the function so you can access it later, you need to define the AEL variable as a global variable, and you must set the variable value prior to creating your new function.This is similar to accessing a parent method in C++, but the functions are not stored in classes.

Example 1: Subclassing an Original AEL Function

You have a function called *cnexOutputTopcellHeader*. You do not want to change how the function works, you just want to add a comment to it. The following code maintains the original function and adds your comment:

```
decl originalCnexOutputTopcellHeader=cnexOutputTopcellHeader;
defun cnexOutputTopcellHeader(designName, dsnH)
{
decl net=originalCnexOutputTopcellHeader(designName, dsnH);
```

```
decl returnNet="* Subclassed the original function to add this comment\n";

returnNet=strcat(returnNet, net);

return(returnNet);
}
```

The first line creates the global AEL variable *originalCnexOutputTopcellHeader* and assigns it the value *cnexOutputTopcellHeader*.

Note The function value is a memory pointer, so the global variable points to the same memory location as *cnexOutputTopcellHeader*. Once you have assigned the variable value, you can call that variable as if you had used a *defun* call to create a brand new function.

The fifth line adds the comment to the value generated by the function to which the variable *originalcnexOutputTopcellHeader* points, the function *cnexOutputTopcellHeader*.

Chapter 5: Setting up GUI Options

The *Netlist Exporter* manual describes the netlist exporting dialog. The dialog has a command button labeled *Modify Option List*. When you click it, a file named *cnexOptions* loads in the location specified by the path in CNEX_EXPORT_FILE_PATH. The *GUI Option* dialog creates this file, the content of which depends on the tool you have chosen. The *cnexOptions* file loaded by the GUI dialog overrides the default version of *cnexOptions* that comes with Front End Flow.

Note A working knowledge of AEL programming is required to setup GUI options.

Option List Global Variable

The Front End Flow API function *cnexExportNetlistHeader* outputs netlist lines at the beginning of the netlist file. The header lines include comment lines, file includes, and global option statements.

The function *cnexExportNetlistHeader* collects the global option statements from the global variable *cnexExportOptionList*. This variable contains one text entry for each option line that appears in the netlist.

The following list shows some netlist options for various tools:

- Dracula: `*.BIPOLAR`
- HSpice: `.TEMP 25`
- ADS: `Options ResourceUsage=yes`

To output an option into the netlist file, the global option variable, *cnexExportOptionList* must have a line that specifies the option.

For example, to get the **.BIPOLAR* option to appear in the header of a Dracula netlist file, write the following line:

```
cnexExportOptionList=list("*.BIPOLAR");
```

This causes a single option, **.BIPOLAR*, to be output in the netlist header.

You can also add more options to the *cnexExportOptionList* variable by using the ADS append command as follows:

```
cnexExportOptionList=append(cnexExportOptionList, list("*.CAPVAL"));
```

This adds the option, **.CAPVAL*, to the list.

The following method is recommended to build up your option list:

1. Read all of your options from a configuration file.
2. Use the append function to build up the final *cnexExportOptionList*.

Option List Global Variable for Dracula

In the *cnexOptions* file, Dracula has a function, *cnexSetupDraculaOptions*, that uses the ADS AEL function *getenv* to retrieve configuration file values. It then checks the values of the configuration variables and determines how to set up the global option variable, *cnexExportOutputList*.

The following code is an example of the use of *cnexSetupDraculaOptions* for conditional loading of options. The Dracula options used depend on the options in the global options list.

Making an Options List for Dracula

```
defun draculaConvertToBoolean(value)
{
   if(value == "1")
return(TRUE);
else
return(FALSE);
}

defun cnexCreateNetlistOptionList(value)
{
  cnexExportOptionList=append(cnexExportOptionList, list(value));
}

defun cnexSetupDraculaOptions()
{
cnexExportOptionList=NULL;
decl bipolar=draculaConvertToBoolean(getenv("bipolar", "dracula"));
if(bipolar)
{
cnexCreateNetlistOptionList("*.BIPOLAR");
decl capa=draculaConvertToBoolean(getenv("capa", "dracula"));
if(capa)
```

The function *getenv(<option>, <file>)* receives the specified option from the specified file name.

The function *draculaConvertToBoolean* checks to see if the value returned for a configuration variable is *1*. If it is, the value returned is the boolean *TRUE* value; otherwise, *FALSE* is returned. This function is needed because the *getenv* function will return strings, even if a value could be interpreted as a number.

Note If you request a configuration variable that does not exist with the *getenv* function, it will return *NULL*. Otherwise, the text value of the variable will be returned.

It is usually recommended that your option configuration file have the same name as the Front End Flow configuration file. However, this is not required because the configuration file name can be hard coded

It is recommended that you write a function that retrieves the options settings from a configuration file. That way, you can set the global options list up by calling the function anywhere within your own code.

Overriding the cnexNetlistDialogOptions_cb Function

Clicking the *Modify Option List* button in the *GUI Option* dialog calls the function *cnexNetlistDialogOptions_cb*. The default of this function that comes with the Front End Flow installation returns the following message:

```
There are no options for the tool <tool>
```

If you want the user to be able to see and modify available options for a tool, you will need to create a new *cnexNetlistDialogOptions_cb* function that provides a dialog window that allows the user to set up options graphically.

Note If you want end users to always use the same options, set up a file that is automatically included (see "Setting Up Automatically Included Files" on page 4-1). Put the options in the file with which you want the end user to work.

Function Prototype

Dialog windows are created in ADS using an extension to the AEL language called Layered API (LAPI). LAPI is a C++ based library that has wrappers to allow you to create dialog windows.

The prototype for the API callback function for dialog options is: *cnexNetlistDialogOptions_cb(<object handle>,<data handle>, <window instance handle>).*

Example of the API callback function for the *Netlist dialog options window*:

cnexNetlistDialogOptions_cb(buttonH, mainDlgH, winInst)

- The first argument passed is the handle of the object that initiated the callback function. For this function, the object is the *Modify Option List* button.
- The second argument in an API callback is a data item. For this function, the handle to the Front End Flow netlist dialog has been passed.
- The third argument for a LAPI callback is a window instance handle. Dialogs are associated to certain windows. In this case, *winInst* will point back to the schematic window that was used to bring up the Front End Flow netlist exporting dialog.

Note You must use this exact protoype.The Front End Flow netlist exporting dialog always calls the function *cnexNetlistDialogOptions_cb* which always uses the same three arguments.

Note To get further information on LAPI, contract Agilent for a special training class with *Agilent Technologies Solution Services* group.

Creating a Dialog Box

The command *api_dlg_create_dialog* creates a new dialog box and returns the dialog handle. The command has eight arguments. You must specify the first two, the remaining six are optional.The optional arguments specify elements of the dialog box.

Example of the Function Call

```
decl dlgH=api_dlg_create_dialog (dlgName, winInst, ...<the six optional
arguments>)
```

Required Arguments

dlgName

This argument assigns the name for the dialog box.

winInst

This argument specifies the window instance. Use the instance passed in the *cnexNetlistDialogOptions* function.

Optional Arguments

API_RN_CAPTION

This specifies the text displayed in the dialog banner. It must be a text value.

Usage: `API_RN_CAPTION, “My Option Dialog”,`

API_RN_ORIENTATION

This specifies whether the dialog is laid out horizontally or vertically. Horizontal layout means that when you specify a new dialog item it is placed next to the prior dialog element. Vertical layout means it is placed below.

A vertical layout is recommended for a dialog.

- Specify *API_RV_VERTICAL* for a vertical layout.
- Specify *API_RV_HORIZONTAL* for a horizontal layout.

Usage: `API_RN_ORIENTATION, API_RV_VERTICAL`

API_RN_DEFAULT_OPTIONS

This allows you to specify what table options the dialog box elements inherit from the parent dialog.

It is recommended to always specify this as *API_RV_TBL_LEFT*.

Usage: `API_RN_DEFAULT_OPTIONS, API_RV_TBL_LEFT,`

API_RN_TBL_OPTIONS

This specifies the table options for the dialog box. Dialog box elements also inherit these options.

It is recommended to use *API_RV_TBL_LK_HEIGHT|API_RV_TBL_SM_HEIGHT*. This forces the dialog to use the smallest height possible, and does not stretch dialog elements to fit the dialog space.

Usage: `API_RN_TBL_OPTIONS,API_RV_TBL_LK_HEIGHT|API_RV_TBL_SM_HEIGHT,`

API_RN_RESIZE_MASK

This specifies the resizing mask.

It is recommended to specify *API_RV_DLG_MIN_WIDTH|API_RV_DLG_MIN_HEIGHT*, so that the dialog is created with the minimum possible width and height.

Usage: `API_RN_RESIZE_MASK,API_RV_DLG_MIN_WIDTH|API_RV_DLG_MIN_HEIGHT,`

API_RN_MODE_TYPE

This specifies whether the dialog is modal or non-modal.

For the Front End Flow options dialog box specify *API_RV_MODAL_DIALOG* so that the dialog box is modal.

Usage: `API_RN_MODE_TYPE, API_RV_MODAL_DIALOG,`

Creating Dialog Box Elements

Dialog box elements are objects. Dialog box elements, or objects, include labels and tables, which can also be used to help make command buttons.The Layered API functions used to create dialogs and dialog elements are described in Appendix B, Layered API Functions.

The command to create dialog elements is the Layered API function *api_dlg_create_item*.

Syntax

api_dig_create_item(name, type, [resource, value, ...], [itemH, ...])

- The first argument is the name of the dialog element. Use a unique name for the dialog so that you can retrieve the element by name in other code.
- The second argument is an enumerated integer that defines the type of dialog component. There is a list of the dialog element types in "Layered API Dialog Elements" on page B-4.
- The last argument space, *[itemH, ...]* is used an option argument used for placing children in the dialog object, if that object supports them. It is not treated in this discussion.
- The third argument space, *[resource, value, ...]*, defines the dialog elements or objects. Some examples of making dialog elements follow.

An Empty Label Dialog Element

To create an empty line label, in order to space a dialog out vertically, the following code could be used:

```
api_dlg_create_item("labelSpace1", API_LABEL_ITEM, API_RN_CAPTION, "")
```

Table Elements

Use table elements when you want to group several elements of a dialog box on the same row, or several elements in the same column.

There are three basic steps to defining table elements using the *api_dlg_create_item* call.

1. Set the orientation of the table elements.

   If you items to appear in the same table row, specify a horizontal orientation by entering *API_RN_ORIENTATION, API_RV_HORIZONTAL*. If you want elements to be in a column, specify a vertical orientation by using *API_RN_ORIENTATION, API_RV_VERTICAL*.

2. Add dialog elements to the table.
3. Put a framing box around the dialog.

   This increases legibility. If you specify *API_RN_CAPTION*, the table will automatically have a framing box drawn around it. If *API_RN_CAPTION* is

empty, you must specify *API_RN_FRAME_VISIBLE, TRUE* to have the framing box drawn.

The following code creates a table that contains *OK* and *Cancel* command buttons at the bottom of the dialog box:

```
api_dlg_create_item ("actTabl", API_TABLE_GROUP,API_RN_ORIENTATION,
API_RV_HORIZONTAL,API_RN_EQUALIZE_ALL, TRUE,API_RN_DEFAULT_OPTIONS,
API_RV_TBL_FIX_SIZE,API_RN_TBL_OPTIONS,API_RV_TBL_FIX_HEIGHT,
pbOkay = api_dlg_create_item ("pbOkay", API_PUSH_BUTTON_ITEM,
API_RN_CAPTION, "OK"),
pbCancel = api_dlg_create_item ("pbCancel", API_PUSH_BUTTON_ITEM,
API_RN_CAPTION, "Cancel")
)
```

Figure 5-1. The Result of the Call to the api_dlg_create_item Function

- The orientation of the table is set to *API_RV_HORIZONTAL*, which causes the two command buttons to be placed on the same row.
- Two buttons are created in the table by making calls to *api_dlg_create_item*.
- The variables *pbOkay* and *pbCancel* are set as the return values of *api_dlg_create_item*.

Adding Callback Functions to Dialog Elements

Callback functions add an action to the dialog elements or objects. For example, if you click on the *OK* button, you want a callback function to be executed that will save the options and dismiss the dialog box.

The command to create dialog elements is the Layered API function *api_dlg_add_callback*.

Syntax

api_dig_add_callback(itemH, functionName, callbackType, callabckData)

Example

This *api_dig_add_callback* function adds action to the *OK* command button object created on page 5-7. The *OK* action saves the options and dismisses the dialog.

```
api_dig_add_callback(pbOkay, “cnexOptionDialogOkay_cb”,
API_ACTIVATE_CALLBACK, list(dlgH, tool))
```

- The first argument is the handle to the dialog element to which you want to add the action. The variable *pbOkay* contains the handle to the *OK* command button item.

  The *pbOkay* handle is the first argument passed passed to the callback function specified in the second argument.

Note You cannot specify the text name of your dialog item as the first parameter. Layered API functions will not automatically search for your dialog element in memory based on the name.

- The second argument is a string value that designates the name of the callback function to be called when the object is activated. In the example it is *cnexOptionDialogOkay_cb*.
- The third argument is an enumerated integer that specifies how the object is activated. Each Layered API dialog element supports different triggers.

  In the example, *API_ACTIVATE_CALLBACK* is specified for the *pbOkay* button. The callback function is activated whenever the *OK* command button is clicked.

Note Refer to “Layered API Dialog Elements” on page B-4 for more information on which triggers are available for certain dialog elements.

- The final argument is a pointer to an AEL data list that is the second argument passed to the callback function.

  The data list contains *dlgH*, which is a handle to the option dialog, and tool, which is a variable that is set when you choose the tool for this dialog. You can retrieve the values in this list with the following code which uses the *nth* function:

```
defun cnexOptionDialogOkay_cb(buttonH, cbData, winInst)
{
  decl dlgH=nth(0, cbData);
```

```
    decl tool=nth(1, cbData);
.
.
.
}
```

Displaying the Dialog

After the dialog has been created, and all of the callback functions have been added to the dialog, the dialog can be displayed using the following command:

api_dlg_manage(dlgH)

This should be the last function call.The dialog box is modal, therefore no commands will be executed after the *api_dlg_manage* function has been called. All callbacks must be set up prior to displaying the dialog.

Closing the Dialog

The options dialog box is modal. A modal dialog box takes the input focus and does not allow you to work on anything else until the dialog box is dismissed.

The command to close the dialog box is the function *api_dlg_unmanage*.

Syntax

api_dlg_unmanage(dlgH)

The one argument, *dlgH*, is the handle to the dialog box to be closed.

The dialog box does not allow you to type in commands, therefore you must set up one or more dialog elements to have callback functions that will call the *api_dlg_unmanage* command. Typically, you use the *OK* and *Cancel* buttons for this.

The following code adds the *close* action to the *Cancel* command button in the dialog elements created in “Table Elements” on page 5-7:

```
api_dlg_add_callback(pbCancel, “cnexOptionDialogCancel_cb”,
API_ACTIVATE_CALLBACK, dlgH);
```

This causes the function *cnexOptionDialogCancel_cb* to be called any time the *Cancel* button is clicked. The data argument passed to the function *cnexOptionDialogCancel_cb* will be the dialog handle.

The definition for the cancel function is then set up as follows:

```
defun cnexOptionDialogCancel_cb(buttonH, dlgH, winInst)
{
api_dlg_unmanage(dlgH);
}
```

When the *api_dlg_unmanage* function is called, the dialog box is closed.

Saving Options to a Configuration File

Once the user has chosen the options they wish to set, perform the following steps:

1. Make certain that the global options list gets set up properly.

   This is discussed in "Option List Global Variable" on page 5-1.

2. Store the settings into a configuration file.

Note If stored in the configuration file, the user will not need to set up the options each time they make a netlist.

Getting the Values of the Dialog Box Elements

You need to be able to get the value of a dialog box element in order to know the state of that element. For example, the value of a text box is the text that has been entered. The value of an option button object, or element, tells you if that button has been selected or not.

There are two parts to getting the value. First, you must get the handle of the object whose value you want. The Layered API function for this is *api_dig_find_item*. Second, after you have the object handle, you must get the value from the object. The Layered API function for this is *api_dig_get_resources*.

Note Once you get the object handle, you can also use it in other commands such as *api_dlg_add_callback*.

The syntax for the function to get the object is as follows:

api_dig_find_item(dlgH, name)

- The first argument, *dlgH*, specifies that you are requesting the object handle.
- The second argument, *name*, specifies the name of that object.

The syntax for the function to retrieve the value is as follows:

api_dig_get_resources(itemH, resource, value)

The first argument, *itemH*, is the object handle.

The second argument, *resource*, an an enumerated integer value that designates the resource value you want to retrieve.

The third argument, *value*, gets the value of the specified resource by going to the address of the value.

Note AEL, like C, uses an ampersand (&) prefix to designate the address of a variable.Thus, to pass the address of a variable to a function, you use the code *&name* and not *name*.

Retrieving the Value of a Dialog Box Element

The following code retrieves the toggle state an option check box named *bipolar*:

```
decl itemVal;
decl bipolarH=api_dlg_find_item(dlgH, "bipolarH");
api_dlg_get_resources(bipolarH, API_RN_TOGGLE_STATE, &itemVal);
```

- In the first line, the *itemVal* variable is declared which will hold the value retrieved from the dialog element.
- In the second line, the handle to the dialog object is retrieved using the *api_dlg_find_item* function.

  When the bipolar check box item was created, it was named *bipolarH*. This is the name used to retrieve the handle to that object. Once the handle to the object has been retrieved, that handle is used to get the value of *API_RN_TOGGLE_STATE* by using the function *api_dlg_get_resources*.
- In the third line, the address of *itemVal* is passed into the function. After the function returns, the variable *itemVal* can be accessed to determine whether the check box was checked or not.

Writing a Value to a Configuration File

You can save retrieved values to a configuration file by using the *setenv* command. This command writes a configuration file in the current working directory. For ADS, the current working directory is always the directory that contains the currently open project.

The *setenv* command has four arguments:

- The first argument is the name of the configuration variable. The name you use must match the name of the object from which you retrieved the value. In the example above, the name is *bipolar*. The argument must be a text string.
- The second argument is the value. In the above example, the value is stored in *itemVal*. This value must be a string.

  Use the *is_string* function to determine if the value is a string.

  If the value is not a string, use the *identity_value* function to convert it to a string.
- The third argument is the name of the configuration file where you want to store the value. This is a text name, and should not include the *.cfg* extension. The extension is added automatically.
- The fourth argument specifies the directory where you want to save the configuration file. However, for options, it is recommended that you save the file in the directory that contains the current object. The file is stored there automatically if you leave out this argument. Therefore, do not fill in this space.

Note If the configuration file does not exist, *setenv* will make it automatically. If a value already exists in the configuration file, the *setenv* command will overwrite it.

Writing a Value to a Configuration File

The code in "Retrieving the Value of a Dialog Box Element" on page 5-12 got the state of the bipolar check box. The code below stores that value in a configuration file using the name *bipolar*.

```
if(is_string(itemVal))
setenv("bipolar", itemVal, "dracula");
else
```

```
setenv(“bipolar”, identify_value(itemVal), "dracula");
```

The first line uses the *is_string* function to see if *itemVal* is a string. In this case it is not because it is a Boolean value, a integer value of either *0* or *1*.

Control branches to the last line. The *setenv* command goes to the object named *bipolar*. The command then reads the value in *itemVal* and, using the *identify_value* command, converts it to a string.

The command sends the value to a configuration file named *dracula*. The command automatically adds the file extension *.cfg* which denotes a configuration file.

Summary

These are the steps to make your own custom options dialog:

1. Create an *cnexOptions.ael* file in a directory appropriate for your tool.
2. Write a function that can retrieve option settings from an ADS configuration file.
3. Write a customized *cnexNetlistDialogOptions_cb* function.
4. Write a function that can save the data in the dialog to an ADS configuration file.
5. Write a function that can close the dialog.

Chapter 6: Hspice Netlister Example

This chapter provides an example for creating a custom dialect from the base netlist format shipped with Front End Flow. The HSpice netlist code configured in this example is shipped with Front End Flow. Many of the functions in Front End Flow are provided in source form to allow you to create your own netlist dialect by modifying some key functions.

To create a custom dialect, perform the following steps:

1. "Creating the New Dialect Directories and Files" on page 6-1.
2. "Modifying the Configuration File as Needed" on page 6-3.
3. "Modifying the Netlisting Functions as Needed" on page 6-4.
4. "Creating Component Definitions" on page 6-11.
5. "Verifying the Netlist" on page 6-22.

Creating the New Dialect Directories and Files

The first step for making a new netlisting dialect is to create the directories so that Front End Flow recognizes the new dialect.

As was noted in "Adding a Tool" on page 1-9, Front End Flow checks the variable CNEX_COMPONENT_PATH to locate the Front End Flow tools. Adding in an HSpice directory in one of the component path directories allows Front End Flow to recognize the new HSpice dialect.

Adding a new dialect requires the following three items:

- The component definitions

  This requires making a component definition directory.

- Creating or overriding AEL definitions

  This requires making a code directory to contain the AEL files.

- Writing a configuration file

  This requires making a new configuration file that sets global netlisting variables different from the default settings.

Making the Component Directory

While the test component definitions are under development, place them in a directory that will only be visible to the developer. To do this, put the definitions in *{%CNEX_HOME_DIR}/components*.

Once the definitions are finished, move them to another directory, such as *{%CNEX_CUSTOM_DIR}/components*, or a design kit directory. You could also add your own directory to *CNEX_COMPONENT_PATH* by editing the *CNEX.cfg* file. This allows you to development in your own directory, for example, *$HOME/development/netlist_exp/components/{%CNEX_TOOL}*.

For this example, an HSpice directory is created in *$HOME/hpeesof/netlist_exp/components* which corresponds to *{%CNEX_HOME_DIR}/components*. Place all of the test component definitions in this directory. When the development is finished, move them to an appropriate central directory.

Creating the Source Code Directory

In addition to creating component definitions, there are functions you must override to make the HSpice netlist dialect work. Put the AEL functions to override in a file in their appropriate home directory. The configuration variable CNEX_COMPONENT_PATH defines where Front End Flow will look for AEL files. That location is *{%CNEX_HOME_DIR}/ael/{%CNEX_TOOL}*. You are developing a *CNEX_TOOL* called *HSpice*. Therefore, you make a directory called *HSpice* at *$HOME/hpeesof/netlist_exp/ael*.

Creating the HSpice Configuration File

In addition to overriding functions and creating component definitions, you must also make a configuration file for HSpice. *CNEX_COMPONENT_PATH* searches for component files with the name type *CNEX_config.{%CNEX_TOOL}*. We have been using home directories to place our development files, so use *{%CNEX_HOME_DIR}/config* for the location for making the new configuration file.

It is easier to modify an existing configuration file than to write a completely new one. Use *CNEX_config.dracula* as the basis for the new HSpice configuration file because Dracula uses HSpice style netlists. Copy *CNEX_config.dracula* from *{%LVS_INSTALL_DIR}/config* to *{%LVS_HOME_DIR}/config/CNEX_config.HSpice*.

This satisfies the need to have a configuration file with the name *CNEX_config.{%CNEX_TOOL}* so that *CNEX_COMPONENT_PATH* can find it.

Modifying the Configuration File as Needed

The netlisting configuration file is discussed in "CNEX_config Configuration File" on page 2-7. Compare each of the configuration variables with the with the documentation of the tool you are supporting to determine which, if any, of the configuration variables need to be modified.

Refer to Chapter 2, Configuration Files and use the following example as a checklist for your custom dialect. This example uses HSpice as the custom dialect.

1. Is HSpice dialect case sensitive?

   No. Set *CASE_INSENSITIVE_OUTPUT* to *TRUE*.

2. Is a component instance separator needed?

   No. Leave it blank.

3. Are there any nodes in ADS that should be set to equivalent nodes in HSpice?

   No. Therefore do not add *EQUIV* lines to the file.

4. Are there any expressions in ADS that are different in HSpice?

   Yes. you need to map the ADS expressions to the corresponding HSpice expressions. For example, the function *ln* is *log* in HSpice, so you map the expression as *EXPRESSION_MAPPING = ln log*. Also, the ADS function *log* is equivalent to HSpice's *log10*. Map that expression as *EXPRESSION_MAPPING = log log10*. Map any other expressions required by the tool you are using.

5. Does HSpice require an expression delimiter?

   Yes. HSpice expressions are required to be enclosed in single quote marks. Set *EXPRESSION_START* to ‘, and also set *EXPRESSION_END* to ‘.

6. Does HSpice have a particular node that is ground?

   Yes. *Node 0* is always ground in HSpice. Set the variable *GROUND* to *0*.

7. What character is used to designate a comment line for HSpice?

   HSpice uses the * character at the start of a line to designate the line as a comment. Set *LINE_COMMENT* to *.

8. What is the continuation character?

   The continuation character in HSpice is +. It is placed at the beginning of the following line. Set the variable *LINE_CONTINUATION_CHARACTER* to +. Because the continuation character must be at the beginning of the following line, set *LINE_CONTINUATION_MODE* to *0*.

9. Does HSpice have a maximum line length?

   HSpice input line can be a maximum of 1024 characters. Set *MAX_LINE_LENGTH* to *1024*.

10. Does HSpice allow numeric node names?

    Yes. However, names that begin with a number ignore any alphabetic characters after the numbers. This is not true in ADS. Therefore, use numeric node names that are prefixed. Set the *NUMERIC_NODE_PREFIX* to *_net* in order to be consistent with ADS.

11. Does HSpice support engineering notation?

    Yes. Since engineering format is easier to read, set *SCALAR_TO_SCIENTIFIC* to *FALSE*.

12. Do the HSpice engineering notations match the ADS engineering notations?

    Not in all cases. HSpice does not list their scaling factors in the documentation. This is something that must be determined by experimentation.

Once all of the configuration variables are set, you have the basis for your custom netlist exporter. Next, you need to customize the functions so they will work properly for your tool.

Modifying the Netlisting Functions as Needed

A good approach to modifying the netlisting functions is to create one component definition file utilizing each function that you are going to modify. This will allow to test each function as you write it.

In "Creating the Source Code Directory" on page 6-2, we specified that *{%CNEX_HOME_DIR}/ael/HSpice* is the development directory. We will now make a new file, *cnexNetlistFunctions.ael*, in that directory. This serves as the customization file for netlisting functions. Once the file is created, it will always be loaded as long as HSpice is selected as the netlisting tool.

The modification process consists of the following steps:

1. “Modifying Instance Functions” on page 6-5.
2. “Modifying Header and Footer Functions” on page 6-9.

To test your component definition file perform the following steps:

1. With ADS running, place a component in the schematic.
2. Create a Front End Flow netlist to test your function.

   The function file is always reloaded each time a netlist is created.
3. Test the function as appropriate for the function, for example, review the netlist or simulate a circuit.

Modifying Instance Functions

Front End Flow provides eight instance netlisting functions.

To determine what the current function outputs and if modification is required, perform the following steps:

1. Place ADS standard components on a schematic and netlist them with the Dracula tool selected.
2. Check the output to determine if modification is required.
3. Determine if the component configuration or the function requires modification.

Note If you are not certain if the component or the function requires modification, modify the configuration first. It is easier to modify components.

Before testing, start ADS and create a new project so that no pre-existing information is used. For the example in this chapter, a new project, *HSpiceSetup_aprj*, is created. Next, a new design is created with the name *test1*.

The cnexNetlistInstance Function

The next step is to find out what the current function exports for a known device and what HSpice needs for that device. For simplicity, the following example uses a capacitor as the template component.

To get an initial HSpice definition for the component, copy the Dracula definition into the HSpice components directory.

1. Place a capacitor, component *C*, in the ADS *test1* schematic.
2. Consult the HSpice documentation. According to the documentation, the following is the general format for an element:

   ```
   elname <node1 node2 ... nodeN> <mname>
   + <pname1=val1> <pname2=val2> <M=val>
   ```

   The following is the specific format for a capacitor:

   ```
   Cxxx n1 n2 <mname> <C=>capacitance <<TC1=>val> <<TC2=>val>
   + <SCALE=val> <IC=val> <M=val> <W=val> <L=val>
   + <DTEMP=val>
   ```

3. Find out what the current function returns. Bring up *netlisting* dialog box, and select HSpice as the tool. Select the *View netlist file when finished* check box, and the create a new netlist by clicking *OK*. The resulting netlist line for the capacitor is as follows:

   ```
   cc1 _net2 _net1 C=1pF
   ```

   This matches the HSpice requirement.

You may want to use more complex components to verify more outputs. However, in this example, the *cnexNetInstance* function output is the same as that required by HSpice. Therefore, for the capacitor component, HSpice does not need an override of the *cnexNetlistInstance* function.

The cnexSubcircuitInstance Function

To test the outputs of this function, first make a subcircuit, then follow the procedure below.

Note This example shows what to do if your first test case does not show needed parameter output information.

Make a subcircuit by placing two ports in *test1* design and connecting them to the capacitor that was placed to test the *cnexNetlistInstance* function.

1. Create a symbol by using **View > Create/Edit Schematic Symbol** from a schematic window.

   This creates a new two port symbol for the *test1* component.

2. Save that design and then create a new design, *test2*.

3. Place one instance of *test1* in the *test2* design.

4. Check the HSpice documentation for the definition of a subcircuit. According to the HSpice manual, the following is the definition for a subcircuit call:

   ```
   Xyyy n1 <n2 n3 ...> subnam <parnam=val ...> <M=val>
   ```

   The subcircuit does not have parameters; therefore, the test will not give you output information. You must add a parameter to the test.

5. Select **File > Design Parameters** from the schematic window.

   The *Design Parameters* dialog will appear.

6. Select the **Parameters** tab, and create a new parameter called *C* with a default value of *1 p*. Set the parameter type to *Capacitance*. Add the parameter and save the design.

7. Go back to the top level, delete the instance of *test1* and place it again on the schematic. It now has a parameter, *C*.

8. Netlist the design. The result for the *test1* instance is as follows:

   ```
   xx1 _net1 _net2 test1 C=1p
   ```

This output matches the HSpice requirements, so you do not need to make any changes.

The cnexGlobalNodeInstance Function

To place a *GlobalNodeInstance*, select the menu option **Insert > Global Node**.

Add a new global node, *g1*, to the global node list, and put a wire label on one of the pins of the *test1* instance.

According to the HSpice manual, global nodes are designated in HSpice by outputting a *.global* directive. Thus, to netlist the *GlobalNodeInstance* correctly, it must create a *.global* option in the HSpice netlist.

After placing the global node and creating a new netlist, the result is as follows:

```
.global g1
```

This matches the HSpice requirement, so you do not need to add any changes for *cnexGlobalNodelInstance* in the custom *cnexNetlistFunctions* file for HSpice.

The cnexVariableInstance Function

1. Place a variable instance by inserting a VAR component.
2. Set up three variables in the VAR component: `C1=2p`, `C2=3p`, and `C3=C1+C2`.

   To create a parameter, the HSpice manual states to use the following syntax:

```
.PARAM <SimpleParam> = <value>
.PARAM <AlgebraicParam> = 'SimpleParam*8.2'
```

3. A netlist is created and gives the following results:

```
.param C1=2p
.param C2=3p
.param C3='C1+C2'
```

This is what HSpice expects. Again, the function does not have to be overridden.

The cnexShortInstance Function

Do not override this function.

It takes the output of multiple nodes and replaces all future occurrences of the nodes with one equivalent node, usually the first node of the component.

Use this function on *tline* components, if you want to use the HSpice transmission line component.

The cnexShortMultiportInstance Function

Do not override this function.

It will take the instance list, match the pairs of nodes to each other, and short-circuit the node pairs. The first node in the pair becomes the name used whenever the second node in the pair is encountered anywhere in the current subcircuit.

The cnexUnknownInstance Function

You do not need to override this function.

This function outputs a comment for a component that does not have an HSpice definition.

Modifying Header and Footer Functions

So far in the example none of the default instance netlisting functions were incorrect for the HSpice netlister. The *cnexNetlistFunctions.ael* file is still empty, except for the comment line that has been placed to indicate that this file is customization for HSpice.

Next are the header functions. These functions create the lines output at the beginning of the netlist, the beginning of the top cell, and the beginning of each subcircuit. The footer functions take care of what is output at the end of the netlist, the end of the top cell, and the end of each subcircuit definition.

The cnexOutputSubcircuitHeader Function

This function returns the subcircuit definition line.

The HSpice manual specifies that a proper definition as follows:

```
.SUBCKT subnam n1 < n2 n3 ...> < parnam=val ...>
```

The *test2* circuit already has a subcircuit placed in it, *test1*. The netlist you generated gives the following for the *test1* subcircuit definition:

```
.subckt test1 _net3 _net1 C=1p
```

The output is correct for HSpice: the *.subckt* was output, the nodes are there, and the parameter definition is correct.

The cnexOutputSubcircuitFooter Function

This function outputs the end of a subcircuit definition.

The HSpice manual specifies the following end of a subcircuit definition:

.ENDS <*SUBNAM*>

The netlist generated from testing using *cnexOutputSubcircuitHeader*, returns the following:

```
.ends test1
```

This is a proper subcircuit ending for HSpice. This function does not need to be changed.

The cnexOutputTopcellHeader Function

The *top cell header* appears at the beginning of the output for the top level circuit. In this example, *test2* is the top cell. For HSpice, nothing needs to be set for a top cell. Components can be placed outside of a subcircuit definition, and HSpice recognizes them as being in the top cell.

To run a test, you need to have simulation directives in the top level. If you look at the netlist, you can see the following subcircuit definition for the top cell:

```
.subckt test2
```

This is valid for Dracula, but not for HSpice. This function needs to be changed so it works for HSpice.

Look in the file *cnexNetlistFunctions.ael* in *{%LVS_INSTALL_DIR}/ael*. The current function definition is as follows:

```
defun cnexOutputTopcellHeader(designName, dsnH)
{
    return(cnexOutputSubcircuitHeader(designName, dsnH));
}
```

The Dracula code calls the *cnexOutputSubcircuitHeader* function so that it creates the top cell as if it were a subcircuit. HSpice does not want any top cell output, so write a new function in the custom *cnexNetlistFunctions.ael* file that looks like this:

```
defun cnexOutputTopcellHeader(designName, dsnH)
{
    return("");
}
```

This function returns an empty string instead of a subcircuit top cell header.

The cnexOutputTopcellFooter Function

The *top cell footer* is output after all of the instance definitions for the top cell have been created. In the example, *test2* subcircuit is the top cell. Checking the netlist file, you see that there is no top cell header for *test2* because of the new *cnexOutputTopcellHeader*. However, there is still an end directive for *test2*. The *cnexOutputTopcellFooter* function must be overridden so that there is no footer.

Examine the original code. The default definition is as follows:

```
defun cnexOutputTopcellFooter(designName, dsnH)
{
    return(cnexOutputSubcircuitFooter(designName, dsnH));
}
```

For this example, put in spacing after the end of the top cell and no end subcircuit definition. To do this, the new function definition becomes the following:

```
defun cnexOutputTopcellFooter(designName, dsnH)
{
    return("\n");
}
```

This overrides the default function so that an empty new line is output at the end of the top cell instances.

The cnexExportNetlistHeader Function

The netlist header function outputs the first lines of the netlist. It takes care of outputting options, including files and includes any comments. The default of this option already supplies the correct output for HSpice. Therefore, you do not need to change it.

The cnexExportNetlistFooter Function

This function is called to output lines that appear at the end of the file. For HSpice, the netlist places an *.end* directive at the end of the file. Anything after the *.end* is treated as comments. The default *cnexExportNetlistFooter* function places an *.end* directive. Therefore, you do not need to change the function.

Creating Component Definitions

Now you should set up all of the components needed for your process and your simulation needs. This chapter deals with components that are delivered with ADS. Your foundry kit components should fall into these categories. Because ADS has hundreds of components. This chapter shows only one example in each component category.

Primitive Components

A primitive component is a component that is netlisted and uses one of the built-in simulator components. It has no hierarchy and does not need a model because the parameters of the component represent all of the information needed to define the component.

This example uses a capacitor as a primitive. Before you start, gather the following information:

- The simulator component used by the component in ADS
- The pin count and order used by the component in ADS
- The parameters that the component has in ADS, and whether they are netlisted or not

The Dracula definition is not identical to the HSpice definition.

You have the following information:

- It netlists as a capacitor.
- The pins are 1 and 2, and the order is not important.
- The parameters are *C*, *Temp*, *Tnom*, *TC1*, *TC2*, *wBV*, *InitCond*, *Model*, *Width*, *Length*, and *_M*.

Consult the HSpice documentation to find the following capacitor primitive definition:

```
Cxxx n1 n2 <mname> <C=>capacitance <<TC1=>val> <<TC2=>val>
+ <SCALE=val> <IC=val> <M=val> <W=val> <L=val>
+ <DTEMP=val>
```

or

```
Cxxx n1 n2 <C=>'equation' <CTYPE=val> <above options...>
```

or a polynomial form:

```
Cxxx n1 n2 POLY c0 c1... <above options...>
```

The second and third definitions do not match the ADS ones. The ADS capacitor is set up to match the first definition.

Now that the target format is known, you can edit the definition. In Chapter 3, Component Definitions, two ways of editing a component definition are discussed, using the GUI and editing the file directly. If you have many components to edit, it is easier to edit the text files directly. The GUI is best if you edit one component at a time.

This chapter uses the text editor approach. Open the file *C.cnex* in a text editor.

Modifying the Component Definition Parameters

Set up the function. For the example in this chapter, based on the functions available and the fact that this is not a subcircuit, *cnexNetlistInstance* is the right function to use.

In HSpice a capacitor device name must be prefixed with a *C*. The *Component_Name* field can also be left unchanged, because it is already set to *C*.

Because this is an ideal capacitor, it does not matter which terminal is negative or positive. Based on the ADS symbol, pin 1 is the negative terminal. If polarity is important, you must change the pin association.

Now, you must find out which parameters to netlist, how to map their names into the proper HSpice names, and if any value mapping needs to be done for the parameters. Base this on reading the manual and comparing the parameters for ADS and HSpice.

Additionally, you need to determine which parameters are important for your design. If temperature is not important, do not output them.

HSpice requires the parameters output in the following order:

Model, C, TC1, TC2, SCALE, IC, M, W, L, and *dtemp*

Of these parameters, all must be explicitly named, except for *Model*, which does not have a name, and *C*, where the name is optional.

ADS does not have an equivalent for the parameter *SCALE*. Discard that parameter. ADS has a parameter, *wBV*, which does not correspond to any HSpice parameter. Do not use it.

The parameter *dtemp*, the difference in component temperature from circuit temperature, is not the same as *Temp*, which is the absolute temperature of the component. You must write a function to output the *dtemp* parameter.

The ADS parameters *InitCond*, *Width*, *Length*, and *_M* have definitions that match HSpice parameters *IC, W, L*, and *M*, but have different names. You must map these names.

The new Parameters line is set to the following:

```
Parameters = Model C TC1 TC2 InitCond _M Width Length Temp
```

Note The parameters line specifies the ADS parameter name, not the HSpice parameter name.

The parameter *Model* should output without <*param name*>= for its value. To do this mapping, a the following parameter name-mapping line is placed:

```
Parameter_Name_Mapping = Model
```

Because there is only a single value, the name *Model* is mapped to an empty string. This means that the function will not output a left hand side for the value.

The parameters *C*, *TC1*, and *TC2* do not need to be mapped. There is nothing is put into the file for them.

The parameters *InitCond*, *_M*, *Width*, *Length*, and T*emp* need to be mapped. The following lines are added to handle these parameters:

```
Parameter_Name_Mapping = InitCond IC
Parameter_Name_Mapping = _M M
Parameter_Name_Mapping = Width W
Parameter_Name_Mapping = Length L
Parameter_Name_Mapping = Temp DTEMP
```

The ADS parameter *Temp* is mapped to *dtemp*, but their values are not identical. In “Adding Value Mapping Functions” on page 4-2, the process for writing a value mapping function is described. In this case, the following code returns the correct value for *dtemp*. The ADS parameter *value* contains the absolute value temperature value placed in ADS. The parameter -*temper* contains the circuit value. If you subtract -*temper* from *value*, you get the differential temperature, the value needed for the HSpice parameter *dtemp*. Set up the function to take the ADS parameter *value*, and return the appropriate HSpice value to go into *dtemp*: as follows

```
defun hspiceModifyTemp(value)
{
    decl returnVal;

    returnVal=strcat("’", value, "-temper’");

    return(returnVal);
}
```

To make the *Temp* function use the value, enter the following line:

```
Parameter_Type_Mapping = Temp hspiceModifyTemp
```

The component must now be netlisted. Open *test1* schematic, which already has a *C* component placed.

However, if a parameter does not have a value, it will not be output. Therefore, it is necessary to set values for all of the parameters so that they are netlisted correctly. So, *C* is set to 1.0pF, *Temp* is set to 27, *TC1* is set to .1, *TC2* is set to .01, *InitCond* is set to 1, *Model* is set to CTest, *Width* is set to 10u, *Length* is set to 10u, and *_M* is set to 2. Then a netlist is created as follows:

```
cc1 _net5 _net4 Ctest C=1pF TC1=0.1 TC2=0.01 IC=1 M=2 W=10um L=10um
DTEMP='27-temper'
```

This matches what the output that HSpice requires.

Components that Access Models

Most active devices and some passive devices use components that contain additional parameters for the instance. These auxiliary parameters can be shared among all of the various instances that are similar. These auxiliary components are called *models*.

Some tools, such as ADS and Spectre, treat the model component as a user defined component. When the instance is netlisted, instead of using a component name, such as *BJT*, they use the model name.

Other tools, like HSpice, specify the component name the same as they do for a primitive. The model is just another parameter.

This is an example of setting up a tool for HSpice. Therefore there is no difference between a component that accesses a model and one that does not.

As another example, here is how to set the ADS *BJT NPN* component. As you would for a primitive, gather the following information:

- The simulator component used by the component in ADS
- The pin count and order used by the component in ADS
- The parameters that the component has in ADS, whether they are netlisted or not

If these devices netlist differently into your tool because of the model, you should also get the following information:

- The name of the model parameter
- The type of models that are valid for the component

For this example, you want to netlist *BJT_NPN*, and make it be a *Gummel Poon BJT NPN* device for HSpice.

According to the HSpice manual, the following is the format for the BJT:

```
Qxxx nc nb ne <ns> mname <area> <OFF> <IC = vbeval,vceval>
+ <M = val> <DTEMP = val>
```

or

```
Qxxx nc nb ne <ns> mname <AREA = area> <AREAB = val>
```

```
+ <AREAC = val> <OFF> <VBE = vbeval> <VCE = vceval> <M = val>
+ <DTEMP = val>
```

For HSpice, *model* is not a distinct element, its a parameter. This means that *cnexNetlistInstance* is fine. *Netlist_Function* is set to *cnexNetlistInstance*.

The component name for an HSpice BJT component is *Q*, whether it is *NPN* or *PNP*. The model designates the implanting type for the component. The *Component_Name* parameter is set to *Q*.

The HSpice pin order in this case is collector, base, emitter, with an optional substrate. The ADS symbol has three unnamed pins; 1, 2, and 3. The ADS symbol graphic shows that pin 1 is the collector, pin 2 is the base, and pin 3 is the emitter. Since we need HSpice's node order to be collector, base, emitter, the *Terminal_Order* variable is set to 1 2 3.

The ADS symbol has the parameters *Model*, *Area*, *Region*, *Temp*, *Mode*, *Noise*, and *_M*. *Mode* specifies whether the device is linear or non-linear. *Noise* specifies whether the device is a noise generation source. HSpice does not have an equivalent to either of these parameters, so they're both dropped.

The rest of the parameters do match HSpice parameters, so the parameters value is set to *Model Area Region _M Temp* to match the order that HSpice specifies in its manual.

In HSpice, the *Model* is output as a value only, so a line is put in to eliminate the parameter name, *Parameter_Name_Mapping = Model*.

The chapter example outputs the left hand side of the area value for readability. Since HSpice is not case sensitive nothing is needed for the area parameter.

Region is mapped to the HSpice parameter that designates whether the device is on or off for DC analysis. ADS allows four settings for this value, *0* means the device is off, *1* means the device is on, *2* means the device is reverse biased, and *3* means the device is saturated. The last two are meaningless to HSpice, so these values need to be mapped. Additionally, HSpice does not want integer values, it wants the value to be a text value *on* or *off*. We need a value mapping function for this parameter.

First, the parameter name is mapped so that it is not be output by creating a line as follows:

Parameter_Name_Mapping = Region

Next, a line is created for the value mapping by placing the following line:

Parameter_Type_Mapping = Region hspiceModifyRegion.

The function *hspiceModifyRegion* must now be created. Copy the value mapping function prototype into the file *cnexNetlistFunctions.ael*. The decision here is what to do with the extra values ADS supports. The default value is that the device is on for both ADS or HSpice. So if the value is empty or NULL, the function will return an empty string. For simplicity, set them so that if the *Region* is *1*, the function will return the value *on*. This yields the following function:

```
defun hspiceModifyRegion(value)
{
    decl returnVal;

    if(!value)
      returnVal="";
    else if(value == 1)
      returnVal="on";
    else
      returnVal="off";

    return(returnVal);
}
```

The *_M* parameter needs to be mapped to the parameter *M*. This is done with the following line:

Parameter_Name_Mapping = _M M

The *Temp* parameter is mapped to *DTEMP*, and a value mapping function is specified for *Temp*, *hspiceModifyTemp*. This time, the function has already been written, so it is a matter of adding the following lines to the file:

```
Parameter_Name_Mapping = Temp DTEMP
Parameter_Type_Mapping = Temp hspiceModifyTemp
```

This completes the component definition. The circuit *test3* has a BJT_NPN component and some basic biasing components around it, such as resistors and capacitors. After setting reasonable values for all of the parameters and netlisting, the instance line for the BJT_NPN component is as follows:

```
qbjt1 _net107 _net108 _net109 BJTM1 Area=1 off M=1 DTEMP='27-temper'
```

This is the correct output for HSpice.

Model Components

A model component is a schematic instance that, when netlisted, becomes a model device that other instances in the circuit can access. IC simulators often use model

components. Those simulators also support netlist fragments, pieces of a netlist include in the final netlist which are available only through library calls or include statements.

The ADS simulator, which does not support netlist fragments.

If you use ADS model components in your circuit, the recommendation is to create netlists for HSpice that contain the models you need. Then set up all of the ADS model components so they netlist using the function *cnexIgnoreInstance*. In "Setting Up Automatically Included Files" on page 4-1, there is a description that shows how to get your netlist fragments included in the final netlist.

Simulation Components

It is usually better to include a file that contains the simulations you wish to perform in HSpice instead of an ADS simulation component. Many ADS simulation components do not map into other simulators.

However, for certain simulations, such as DC, you can set up a simulation. The following is an example of setting up a DC component to netlist for HSpice.

Since we know what the component is, check to see what HSpice needs in order to designate a simulation.

When a DC simulation is done, you are trying to find the operating point of the circuit at the time index of zero. To do this in HSpice, the correct line is as follows:

```
.OP <format> <time> <format> <time>
```

Additionally, to perform variable sweeps, you need a DC line as follows:

```
.DC var1 start1 stop1 incr1 <var2 start2 stop2 incr2 >
```

In ADS, the operating point calculation and the variable sweeps are both potentially designated in a single DC component. This is a case of needing two lines of output for a single component.

The only way to figure out the right parameter name is to look at the netlist, and then use the simulator's help capability.

First, generate an ADS netlist, and identify the components line by looking for its instance name. The string in front of the instance name is the device that the component is netlisted as, in this case, DC. To get help on the DC component from the simulator, type in *hpeesofsim -help DC* in a command line terminal. This will give you

the parameters that are valid for the DC device, and a brief description of each parameter.

For the DC operating point, the parameter name in the simulator is *DevOpPtLeve*l.

If *DevOpPtLevel* is placed in the component definition, it will be possible to retrieve its value using the function *cnexGetParameterValues*. The value is examined, and either *NONE*, *BRIEF*, or *ALL* is output, based on the value that was returned. If there was no value, nothing is output at all.

The sweep line is determined by looking at the value of *SweepVar*, which will specify whether a *.DC* sweep will need to be output.

The sweep plan has the variable names *Start*, *Stop*, and *Step*. Assume these are the right parameter names, and set these up on the parameters line along with *SweepVar*. The function will then have to step through and grab these values from the parameter list that was returned from *cnexGetParameterValues*.

Since there is a known set of values set, a *while loop* is set up to output the remaining three parameters. This yields the following component definition file for *DC.cnex*:

```
Netlist_Function = hspiceOutputDcComponent
Component_Name =
Terminal_Order =
Parameters = DevOpPtLevel SweepVar Start Stop Step
```

And the following function definition was created for hspiceOutputDcComponent:

```
defun hspiceOutputDcComponent(instH, instRecord)
{
  /* This is a function that will specifically output a .OP and .DC
     line for HSpice from a DC component. */

  decl net=".OP";

  decl paramList=cnexGetParameterValues(instH, instRecord);
  decl paramRecord, paramValue;

  /* Get the record for DevOpPtLevel */
  paramRecord=car(paramList);
  paramValue=nth(1, paramRecord);

  paramList=cdr(paramList);
  if(paramValue)
  {
    if(paramValue == "0")
    {
      net=strcat(net, " NONE");
```

```
    }
    else if (paramValue == "2")
    {
      net=strcat(net, " BRIEF");
    }
    else
    {
      net=strcat(net, " ALL");
    }
  }

  /* Get the record for SweepVar */
  paramRecord=car(paramList);
  paramList=cdr(paramList);

  paramValue=nth(1, paramRecord);

  if(paramValue)
  {
    net=strcat(net, "\n.DC ", paramValue);
    while(paramList)
    {
      paramRecord=car(paramList);
      paramList=cdr(paramList);
      net=strcat(net, " ", nth(1, paramRecord));
    }
  }

  return(net);
}
```

The component is netlisted, and the result is as follows:

```
.OP ALL
.DC “X” 1000 10000 1000
```

After the first iteration, it appears that the name of the variable is quoted. The parameter formatting function did not take the double quotes out, and ADS specifies the value is explicitly a string.

Two things could be done. A parameter type mapping function could be specified that would remove the quotes, or, code could be added directly into the function to remove the quotes. The second choice has been made in this case, so two new line are added prior to the *net=strcat(net, "\n.DC ", paramValue);* line:

```
if(leftstr(paramValue, 1) == "'")
      paramValue=midstr(paramValue, 1, strlen(paramValue)-2);
```

A DC component is placed, and a netlist is created. Now the output is as follows:

```
.OP ALL
.DC X 1000 10000 1000
```

This is what s needed. It is now possible to run a basic DC simulation in both ADS and HSpice by placing a DC component.

Similar setups could be done for the AC component and the Tran component. Other than DC, AC, and Transient simulation, ADS and HSpice don't have much in common in the way of simulation. These three simulations should be enough to drive model comparison simulations.

Components that Access Netlist Fragment Subcircuits

A netlist fragment is a piece of a netlist that is meant to be reused in other netlists by using library statements or include statements. These can either be models, or they can be complete subcircuits, or even complete subcircuit hierarchies.

If you have a component that is hierarchically defined in ADS, it is a subcircuit and uses *cnexSubcircuitInstance*. If your component is going to access another subcircuit, and it is not hierarchically defined in ADS, but the instance line still needs to be output as a subcircuit reference, you still need to use the function *cnexSubcircuitInstance*. If you use *cnexNetlistInstance*, assuming that because your subcircuit in the fragment is now a new primitive, like it is in ADS, you will not get the correct HSpice format.

For HSpice, a subcircuit is referenced by an instance by using a line with the following format:

```
Xyyy n1 <n2 n3 ...> subnam <parnam=val ...> <M=val>
```

A new component, *test4* is created. This component access one of the following two pre-made netlist fragments that has the subcircuit headers:

```
.subckt cktA pos neg Width=2u Lenth=10u
.subckt cktB pos net Width=2u Length=10u
```

These represent the resistors of two different types. The user chooses between the two resistors by selecting from a pull-down menu on a parameter called *circuit*. This is the type of setup used if you have a high impedance and a low impedance resistor and have parasitic subnetworks to represent each of the two types of resistor where the parasitic values cannot be simply calculated based on parameters that are passed into the circuit.

You can make a component definition for a user defined device. Because the default behavior will not be correct in this case, a new file, *test4.cnex*, is made in the HSpice component directory.

Since the netlist fragments are subcircuits, the *Netlist_Function* is set up to be *cnexSubcircuitInstance*.

The terminal order can be determined from the subcircuit headers. It must be *pos neg*.

You want the subcircuit name, *Component_Name* field, to be picked up from the value that is specified in the circuit parameter. Instead of putting an explicit name, the *Component_Name field* is set to *@circuit*, which tells the netlisting code to use the value of the circuit parameter.

Because circuit is being used as the component name, it does not need to be output as a parameter. The only two parameters are *Width* and *Length*. For this particular subcircuit, the parameter names in the SPICE file are the same as the parameter names of the component. No name mapping is required. The final component definition file becomes the following:

```
Netlist_Function = cnexSubcircuitInstance
Component_Name = @circuit
Terminal_Order = POS NEG
Parameters = Width Length
```

Two instances of *test4* are placed in a new circuit, *test5*. Once instance has circuit set to *cktA* and the other has circuit set to *cktB*. A netlist is generated and the output lines are the following:

```
xx2 _net28 _net27 cktb Width=2uM Length=10uM
xx1 _net28 _net27 ckta Width=2uM Length=10uM
```

These two instance lines match the needed output for the subcircuit headers that were shown.

Verifying the Netlist

Verifying the netlist comprises of making sure that the subcircuit definitions are output correctly and that each instance is output correctly.

For the HSpice simulator ready netlists, to back annotating the DC results to the ADS schematic, you can name all of the nodes in the schematic, which will force ADS to store the DC results into a dataset file. You can then view the results of the ADS simulation in the Data Display Server, and the HSpice results in their results viewer.

If you have a schematic in another tool that can drive HSpice, you can create a netlist from that tool, and a netlist from ADS, and view simulation results from both of the netlists.

Component Verification

Here is a check list to follow that will allow you to verify any component:

1. Determine the ADS component type.
2. Determine the ADS terminal order.
3. Find out the ADS component parameters.
4. Determine the format needed by the new tool. For example, to make a capacitor in HSpice, the format is the following:

   `Cxxx n1 n2 <`*mname*`> <C=>capacitance <<TC1=>`*val*`> <<TC2=>`*val*`> <SCALE=`*val*`> <IC=`*val*`> <M=`*val*`> <W=`*val*`> <L=`*val*`> <DTEMP=`*val*`>.`

5. Create a component definition for the ADS component.
6. Place one instance of the component in a schematic. Make sure to set all of the parameters so they have values.

   This will guarantee that parameters that are supposed to be netlisted are netlisted, and that parameters that aren't supposed to be netlisted are not.

7. Create a netlist. Make sure to set the checkbox so that the netlist will be shown after netlisting is finished.
8. Compare the instance line that was output to the format line that you determined was needed.

   If they match, you are finished. If they do not match, determine if it is because you need a value mapping function, or if you mis-configured something. Also, consider whether your format may need to have configuration variables or the instance function itself changed.

Appendix A: Front End Flow Functions

This appendix contains a listing of the default Front End Flow functions. Every entry has a description of the function, its arguments, and its return valued. You can override any of these functions unless specifically noted. But it is critical that any function that is overridden must have the same argument list and the same return value name.

Instance Netlist Exporting Functions

These functions are generic functions that can format an instance for a netlist. If you are using a tool other than Dracula, Calibre, or Assura, you may need to override them in order to support your tool. All of these functions are provided in source form in the file *cnexNetlistFunctions.ael*.

cnexGlobalNodeInstance

This function reads an ADS global node instance and returns a string with the appropriate syntax for a global node statement.

Syntax

`cnexGlobalNodeInstance(`*instH, instRecord*`);`

Where

instH is the handle to the instance

instRecord is a list of lists read from the component definition file

cnexVariableInstance

This function reads an ADS VAR component instance, and returns a string with the appropriate variable definition format.

Syntax

`cnexVariableInstance(`*instH, instRecord*`);`

Where

instH is the handle to the instance

instRecord is a list of lists read from the component definition file

cnexUnknownInstance

Do not override this function. Any time an instance is determined to be a primitive, and it does not have a component definition file for the current tool, this function will be called automatically. This function will output a warning to the log file, and a comment line into the netlist file. If you have a component that calls this function, it means you want to ignore the component. By using this function, the netlist can still be created without requiring extra work to be done.

Syntax

`cnexUnknownInstance(`*instH, instRecord*`);`

Where

*instH*Handle to the instance

*instRecord*List of lists read from the component definition file

cnexIgnoreInstance

The *cnexIgnoreInstance* function bypasses the processing of an instance. For components that are attached in a schematic, this results in an open circuit at the point where the component is connected. Use the *cnexIgnoreInstance* function with detached components, such as model components, and with parasitic components that are placed in parallel with other components, such as parasitic capacitors. The return value is an empty string.

Syntax

`cnexIgnoreInstance(`*instH, instRecord*`);`

Where

instH is the handle to the instance

instRecord is a list of lists read from the component definition file

cnexShortInstance

The *cnexShortInstance* function shorts all of the nodes of a device together into a single node. These nodes are collected into a global list which is used to replace all of the nodes when the instances that are not shorted are finally output. Use this function to short circuit transmission line components, such as the *mlin* or *tee*, or to short circuit parasitic devices that are connected in series. This function is called internally and should not be overridden. The return value is an empty string.

Syntax

`cnexShortInstance(`*instH, instRecord*`);`

Where

instH is the handle to the instance

instRecord is a list of lists read from the component definition file

cnexShortMultiportInstance

The *cnexShortMultiportInstance* functions shorts pairs of pins on a device together into a single node. These nodes are collected into a global list which is used to replace shorted node names when other instances are output. Use this function to short circuit multi-port transmission lines, such as the four port coaxial cable. This function is called internally and should not be overridden. The return value is an empty string.

Syntax

`cnexShortMultiportInstance(`*instH, instRecord*`);`

Where

instH is the handle to the instance

instRecord is a list of lists read from the component definition file

cnexNetlistInstance

This is the generic function for outputting instances as primitives. If you use a tool other than Dracula, Calibre, or Assura, you must override this function so that it supplies the correct output for your tool. The return value is a string that represents a component for a particular tool.

Syntax

`cnexNetlistInstance(`*instH, instRecord*`);`

Where

instH is the handle to the instance

instRecord is a list of lists read from the component definition file

cnexSubcircuitInstance

This is the generic function for outputting instances that represent hierarchical subcircuits. If you use a tool other than Dracula, Calibre, or Assura, you must override this function so that it supplies the correct output for your tool. The return value is a string that represents a subcircuit call for a particular tool.

Syntax

`cnexSubcircuitInstance(`*instH, instRecord*`);`

Where

instH is the handle to the instance

instRecord is a list of lists read from the component definition file

Subcircuit Header Functions

These functions output the header for a hierarchical subcircuit in a netlist.If you use a tool other than Dracula, Calibre, or Assura, you must override these functions so that they supply the correct output for your tool. Both functions are provided in source form in the file *cnexNetlistFunctions.ael*.

cnexOutputSubcircuitHeader

This function formats the header for a subcircuit. The default function will return an Hspice syntax subcircuit statement. The function also outputs the line to the netlist, so it is not necessary to also call the *cnexExportWriteToNetlist* function. If you use a tool other than Dracula, Calibre, or Assura, you must override this function so that it supplies the correct output for your tool.

Syntax

`cnexOutputSubcircuitHeader(`*designName, dsnH*`);`

Where

designName is a string containing the design name

dsnH is the handle of the design

Example

```
decl dsnH=db_get_design(“myDesign”);
decl net=cnexOutputSubcircuitHeader(“myDesign”, dsnH);
fputs(stderr, net);
```

.subckt myDesign in out

cnexOutputTopcellHeader

This function formats the header for the top cell circuit. The default function will return an HSpice syntax subcircuit statement which is ready to use with Dracula. If you use a tool other than Dracula, calibre, or Assura, you must override this function so that it supplies the correct output for your tool.

Syntax

`cnexOutputTopcellHeader(`*designName, dsnH*`);`

Where

designName is a string containing the top cell design name

dsnH is the handle to the top cell

Subcircuit Footer Functions

These functions are responsible for writing out hierarchical subcircuit footers, for example, the *.ends* statement in a spice netlist. If you use a tool other than Dracula, calibre, or Assura, you must override these functions so that they supply the correct output for your tool. Both functions are provided in source form in the file cnexNetlistFunctions.ael.

cnexOutputSubcircuitFooter

This function formats the footer for a subcircuit. The default function will return an HSpice syntax subcircuit *.ends* directive, which signals the end of the subcircuit. If you use a tool other than Dracula, calibre, or Assura, you must override this function so that it supplies the correct output for your tool.

Syntax

`cnexOutputTopcellFooter(`*designName, dsnH*`);`

Where

designName is a string containing the top cell design name

dsnH is the handle to the top cell

Example

```
decl dsnH=db_get_design(“myDesign”);
decl net=cnexOutputSubcircuitFooter(“myDesign”, dsnH);
fputs(stderr, net);

.ends myDesign
```

cnexOutputTopcellFooter

This function formats the footer for the top cell circuit. The default function will return an HSpice syntax subcircuit end directive, which is appropriate for Dracula. If you use a tool other than Dracula, calibre, or Assura, you must override this function so that it supplies the correct output for your tool.

Syntax

`cnexOutputTopcellFooter(`*designName*`, `*dsnH*`);`

Where

designName is a string containing the top cell design name

dsnH is the handle to the top cell

Netlist Header Function

cnexExportNetlistHeader

This function returns a string that outputs the first lines in the netlist file. The string can contain new line characters to make it span more than a single line of output. This function is always the first function called when generating a netlist. The default function will output global nodes, option statements, comments, and include files.

Syntax

`cnexExportNetlistHeader(`*designName*`, `*repH*`);`

Where

designName is a string containing the top cell design name

repH is the handle to the schematic representation of the top cell

Netlist Footer Function

cnexExportNetlistFooter

This function outputs the ending lines of the netlist. The default function will output an HSpice *.end* directive at the end of the netlist.

Syntax

`cnexExportNetlistFooter(`*designName*`, `*repH*`);`

Where

designName is a string containing the top cell design name

repH is the handle to the schematic representation of the top cell

Circuit Output Functions

cnexOutputSubcircuit

This is a wrapper function that makes all of the calls necessary to completely output a subcircuit. The function calls *cnexExportSubcircuitHeader*, *cnexOutputCircuitData*, and *cnexExportSubcircuitFooter*. The function will always return a NULL.

Syntax

`cnexOutputSubcircuit(`*designName*`);`

Where

designName is a string containing the name of the subcircuit

cnexOutputCircuitData

This function examines all of the instances in a schematic representation and calls the appropriate output functions for each of the instances. The function has no return value.

Syntax

`cnexOutputSubcircuit(`*designName*`);`

Where

repH is a handle to the schematic that will be output

topLevel is a boolean value designating whether the current representation is the top level circuit or not

Parameter Formatting Functions

These functions reformat a parameters value so it can be output into a netlist. These functions are called internally, so it is critical that the functions return the correct values.

cnexExportFormatValue

This function takes a string value with valid ADS metric or english scalars and units, and returns a value that has a number with english or metric scalars appropriate for the netlist tool. The function uses replacement values from the configuration file found in the path location specified in the variable *SCALAR_UNIT_MAPPING*.

Syntax

`cnexExportFormatValue(`*val*`);`

Where

val is a parameter value that needs to be formatted for a particular tool.

Example

```
cnexExportFormatValue(“900 MHz”);
```

returns *900meg*

cnexExportScalarExpand

This function takes a string value that represents a number in metric or english scalars and converts it to a scientific notation string.

Syntax

`cnexExportScalarExpand(`*val*`);`

Where

val is a parameter value that is to be converted to scientific notation

Example

```
cnexExportScalarExpand(“1mOhm”);
```

returns *1e-3*

Global Variable Functions

cnexExportReadGlobals

This function is called prior to exporting a Front End Flow netlist. It reads the *CNEX_config* file for the current tool. This function definition is in *cnexGlobals.ael*, and you can override if it is necessary to write your own variables into a *CNEX_config* configuration file for a custom tool. It is not necessary to call this function in any user defined code.

Syntax

```
cnexExportReadGlobals();
```

cnexExportClearGlobals

This function is called prior to outputting a netlist and directly after a netlist is created. The function sets all lists and string values that have been defined back to their default values or to NULL to conserve memory. This function is in *cnexGlobals.ael*. Override this function if you add your own new global variables.

Syntax

```
cnexExportClearGloblas();
```

Option Functions

cnexNetlistDialogOptions_cb

The function creates a dialog box that contains the options that have been set up for your tool. You must override the *cnexNetlistDialogOptions_cb* function for your tool if you wish to allow the user to graphically specify option settings for a netlist. This function is provided in source form in the file *cnexOptions.ael*. For more information on how to set this function up, see Chapter 5, Setting up GUI Options.

Syntax

```
cnexNetlistDialogOptions_cb(buttonH, mainDlgH, winInst);
```

Where

buttonH is the handle to the button that was clicked to initiate the function call

mainDlgH is the handle to the dialog that initiated the function call

winInst is the handle to the window that the dialog belongs to

Core Functions

The core functions should not be overridden. These functions access the database directly. They also provide extra intelligence that modifies the return values from the database to include effects from power pins, instance iteration, and bus vector notation. These functions should be called in your own functions.

Note If you do find that you must override one of these functions, you must contact Agilent Technologies directly to request the source code. The source code for these functions is not provided in any of the AEL functions distributed with Front End Flow.

---

app_add_cnex_menus

Add this function to *USER_MENU_FUNCTION_LIST* in order to have the Front End Flow menu show up on the tools menu of a schematic window. It should not be called directly. It is meant to be called when a window is being created.

Syntax

`app_add_cnex_menus (`*winTempId*`);`

Where

winTempId is an enumerated integer that designates the type of window to add the menu to. This function will only work if *winTempId* is *SCHEMATIC_WINDOW*

cnex_bound

The *cnex_bound* function uses the *on_error* function to redirect errors that result from using an undeclared variable or an undeclared function. The redirected error function will return NULL if an attempt to reference an undeclared function or variable is made. If the variable or function exists, the value of the variable or function is returned. This function can be used to see if functions that were declared in other ADS modules have been loaded prior to using them. It can also verify if global variables declared in other ADS modules have been defined prior to using them.

Syntax

```
cnex_bound(var);
```

Where

var is the string name of the variable or function to check

Example

```
decl x=1;
cnex_bound(“x”);Returns 1
cnex_bound(“y”);Returns NULL
```

cnexExpandBusNotation

This is a general purpose function that returns an expanded list of all of the items for bus notations. This function can handle any notation that Cadence DFII bus vector notation supports. Therefore, this function is appropriate for any instance, pin, or wire that uses bus notation. That means it can expand more than ADS will allow. However, the function works with the smaller ADS environment.

Node Name	Node List
a	a
a,b	a,b
<*2>a	a,a
<*2>a,<*2>b	a,a,b,b
<*2>(a,b)	a,b,a,b
a<0>	a<0>
a<0,1,2>	a<0>,a<1>,a<2>
a<0:1>	a<0>,a<1>
a<0:1>,b	a<0>,a<1>,b
a<0:2:2>	a<0>,a<2>
a<0:1*2>	a<0>,a<0>,a<1>,a<1>
a<(0:1)*2>	a<0>,a<1>,a<0>,a<1>

Syntax

```
cnexExpandBusNotation(busString);
```

Where

busString is the bus vector string that is to be expanded

Example

```
cnexExpandBusNotation(“a,b”);Returns list(“a”,”b”)
cnexExpandBusNotation(“a<0:2>”);Returns list(“a<0>”,”a<1>”,”a<2>”)
```

cnexExport

This function generates a netlist. The Front End Flow netlist dialog box calls this function directly, and allows you to set global variables prior to calling this function. If you wish to make your own non-interactive functions that will create non-ADS netlists, this function should be called.

Syntax

```
cnexExport( cnexType, designName, netlistName, logName );
```

Where

cnexType is the name of the tool to generate the netlist for

designName is the name of the top cell design to export

netlistName is the full path name of the netlist to produce

logName is the full path name of a log file for error and warning messages

cnexExportAsciiCode

This function returns an ASCII number for a character. Use it to replace illegal characters with a numeric code in the function *cnexExportFixIllegalChars*. If a character is not found, the code *45* is returned.

Syntax

```
cnexExportAsciiCode(c);
```

Where

c is the character to get the ASCII code for

Example

```
cnexExportAsciiCode(“a”);Returns 65
```

cnexExportFindAllSubcircuits

This is a recursive function examines the hierarchy of the schematic representation that was passed to it and looks for all instances that have hierarchy. The function

returns a list of design names that are subcircuits. The function additionally checks the node names of each representation and adds any node that ends in the character *!* to the global node list.

Syntax

`cnexExportFindAllSubcircuits(`*repH*`);`

Where

repH is the schematic representation to search for hierarchical components

cnexExportFixIllegalChars

This function examines a string value and replaces all of illegal characters with the appropriate ASCII code. The ASCII code is prefixed and suffixed with an underscore character to designate that it was an illegal character code and not simply a number that was part of the name. The function returns the fixed string value.

Syntax

`cnexExportFixIllegalChars(`*val*`, `*charList*`);`

Where

val is the string value that needs to have illegal characters replaced

charList is the string of characters that are illegal

Example

`cnexExportFixIllegalChars(“abc”, “b”);Returns “a_66_c”`

cnexExportItemdefParmAttribute

This routine retrieves the attributes associated with a parameter in an ADS item definition. It returns the integer value of the attribute.

Syntax

`cnexExportItemdefParmAttribute(`*parmDefH*`, `*parmName*`);`

Where

parmDefH is the handle to the head of the item definition parameter list

parmName is the name of the parameter to retrieve an attribute for

cnexExportWriteToLog

This function writes the text passed in to it out to the log file that was specified in the *cnexExport* function call. The text string passed in to the function may contain new line characters to force output to be more than a single line.

Syntax

```
cnexExportWriteToLog( text );
```

Where

text is the text to output to the log file

cnexExportWriteToNetlist

This function writes the text string passed to it out to the netlist file that was specified in the *cnexExport* function call. The text string is processed so that it does not exceed the maximum line length and has the appropriate continuation characters added at the start or end of the current line.

Syntax

```
cnexExportWriteToNetlist( net );
```

Where

net is the string to output to the netlist file

cnexGetComponentName

This function gets the component name appropriate for the current tool. The instance component definition is consulted to see if a specific component name has been defined. If it has, that component name will be used.

Otherwise, the function checks to see if the component name field is empty or is a subcircuit, and if the function name for output is *cnexSubcircuitInstance*. If the component is a subcircuit, the ADS component name is used. If component name is empty and the function is not *cnexSubcircuitInstance*, the component name is returned as an empty string.

Syntax

```
cnexGetComponentName(instH, instRecord);
```

Where

instH is the instance handle of the component

instRecord is the component definition record for the instance

cnexGetInstanceName

This function returns the instance name to use for the current instance handle. In cases where a component is being iterated because of bus notation, the instance name is returned with the appropriate iteration counter.

Syntax

`cnexGetInstanceName(`*instH*`);`

Where

instH is the instance handle of the instance to get the instance name from

cnexGetInstanceRecord

This function searches the component definition path and reads in the appropriate component definition file for the specified instance or design name. If the instance handle is specified, the design name is retrieved from the instance handle. If a specific definition directory is specified, and that directory contains a definition for the component, that definition is read instead of searching the component definition path. If the definition directory is specified, and definition exists in that directory, the component path will be searched instead.

If no component definition record can be found for the instance, a default record is created. The default record checks to see if the component is a subcircuit definition. If it is, the default definition is set up to output a subcircuit record, using *cnexSubcircuitInstance*. If the component is not a subcircuit, it is set up to use *cnexUnknownInstance* instead.

Syntax

`cnexGetInstanceRecord(instH, [`*dsnName*`, `*definitionDir*`]);`

Where

instH is the current instance to get the definition record for

dsnName is the design name to get a record for if instH is NULL

definitionDir is an optional directory to read the component definition file from

cnexGetInstances

This function retrieves a list of the instances for a representation. The return value is a list of instance handles for the schematic representation.

Syntax

`cnexGetInstances(`*repH*`);`

Where

repH is the schematic representation to get the instance list for

cnexGetParameterList

This function retrieves the parameter list from an instance record.

Syntax

`cnexGetParameterList(`*instRecord*`);`

Where

instRecord is a list that represents the data from the component definition file

cnexGetParameterValues

This function retrieves the values set for an instance for all of the parameters that are in the instance record parameter list. The list records are the mapped name, the parameter value, and the original parameter name. The function returns a list of lists.

Syntax

`cnexGetParameterValues(`*instH, instRecord*`);`

Where

instH is the instance handle of the current instance

instRecord is a list that represents the data from the component definition file

cnexGetPinConnections

This function looks for the instance handle and the terminal order and returns a list of nodes for connectivity with the instance. The list is ordered to match the terminal order of the instance. This function takes into account bus vector notation and return node names that have the proper bus indices on them.

Syntax

`cnexGetPinConnections(`*instH, instRecord*`);`

Where

instH is the instance handle of the current instance

instRecord is a list that represents the data from the component definition file

cnexGetTerminalOrder

This function returns a list that represents the instance terminal order. The order is determined by looking at the *termOrder* field of the *instRecord*. If the *termOrder* field is not set, the instance symbol is consulted, and the terminal order is determined by reading the pin number from each of the symbol pins. This function also sets up the global variable *cnexInheritedConnectionList* based on whether power pins have been set up on the instance or in the hierarchy of the instance.

Syntax

`cnexGetTerminalOrder(`*instH, instRecord*`);`

Where

instH is the instance handle of the current instance

instRecord is a list that represents the data from the component definition file

cnexGetCustomDir

This function reads the Front End Flow configuration files and returns the value set for *CNEX_CUSTOM_DIR*. If the value is not set in a configuration file, it returns *$HPEESOF_DIR/custom/netlist_exp*.

Syntax

```
cnexGetCustomDir();
```

cnexGetHomeDir

This function reads the Front End Flow configuration files and returns the value set for *CNEX_HOME_DIR*. If the value is not set in a configuration file, it returns *$HOME/hpeesof/netlist_exp*.

Syntax

```
cnexGetHomeDir();
```

cnexGetInstallDir

This function reads the Front End Flow configuration files and returns the value set for *CNEX_INSTALL_DIR*. If the value is not set in a configuration file, it returns *$HPEESOF_DIR/netlist_exp*.

Syntax

```
cnexGetInstallDir();
```

cnexGetTool

This function reads the Front End Flow configuration files and returns the value set for *CNEX_TOOL*. If the value is not set in a configuration file, it returns *dracula*.

Syntax

```
cnexGetTool();
```

cnexGetToolList

This function reads the configuration files, and retrieves the value of *CNEX_COMPONENT_PATH*. It uses that value to search for which tool directories exist in the current component path. It then returns a list which contains the available tools for Front End Flow.

Appendix B: Layered API Functions

This appendix contains an abbreviated list of Layered API functions. The information in this appendix is provide to help in the creation of option dialogs boxes. The functions discussed are those used to create the default options dialog box and those used to generate the Dracula options dialog box.

This appendix also contains a listing of the dialog elements that have been used in the default options dialog and the Dracula options dialog.

Layered API Functions

The following lists the layered API functions and definitions.

api_dlg_add_callback

This function adds a callback to the dialog object. The function does not return a value.

Syntax

`api_add_callback(`*itemH*`, `*functionName*`, `*callbackType*`, `*callbackData*`);`

Where

itemH is the handle to the dialog object

functionName is a string designating the name of the callback function

callbackType is an enumerated integer that designates the action that will cause the callback to be executed

callbackData is a pointer to the AEL data that will be passed as the second argument to the callback function

Example

```
api_dlg_add_callback(pbClose, “close_cb”, API_ACTIVATE_CALLBACK, dlgH);
```

api_dlg_create_dialog

This function creates a custom dialog box according to the specified arguments. The function returns the handle to the dialog.

Syntax

`api_dlg_create_dialog(`*dlgName*`, `*winInst*`, [`*resource*`, `*value*`, ...], [`*itemH*`, ...]);`

Where

dlgName is a string name that designates the unique name of the dialog

winInst is the window instance that the dialog is associated to

resource, value These are integer values that designate properties for the dialog box. These allow you to set the dialog caption, the modality, etc. The format specifies the resource type and the value to use for that resource type. You can specify as many resources as you need.

itemH is a handle to a dialog object that will be displayed as a child object in the dialog window. You can specify as many child object as needed. Child objects are created using the command *api_dlg_create_item*.

api_dlg_create_item

This function creates a new dialog object. The function returns the handle to the child object.

Syntax

`api_dlg_create_item( `*name*`, `*type*`, [`*resource*`, `*value*`, ...], [`*itemH*`, ...]);`

Where

name is the name of the dialog object. This should be a unique name, so that the handle to the object can be retrieved using the command *api_dlg_find_item*.

type is an enumerated integer that specifies the type of dialog object will be created.

resource,value is a set of integers and values that specify properties for the dialog object. Each dialog object type has a different set of resources that can be set for it.

itemH is a set of optional dialog objects to place as children in the dialog object, if the dialog object supports having children. For instance, *API_TABLE_GROUP* can have dialog object arguments because it supports having children.

Example

```
pbOkay=api_dlg_create_item(“pbOkay”, API_PUSH_BUTTON_ITEM,
API_RN_CAPTION, “OK”);
```

api_dlg_find_item

This function retrieves the dialog box object with a specified name from its parent. The function returns the handle of the dialog object if it is found, otherwise it returns NULL.

Syntax

```
api_dlg_find_item(parentH, name);
```

Where

parentH is the handle to the parent dialog object

name is the name of the dialog object to find

Example

```
decl pbOkay=api_dlg_find_item(dlgH, “pbOkay”);
```

api_dlg_get_resources

This function gets the value of a specified dialog object resource. The function does not return a value.

Syntax

```
api_dlg_get_resources(itemH, resource, &var);
```

Where

itemH is the dialog object handle from which to retrieve data

resource is an enumerated integer value that designates the resource value to retrieve

&var is the address of a variable that has its value assigned to the resource value

Examples

```
decl val1;
api_dlg_get_resources(checkH, API_RN_TOGGLE_STATE, &val1);
fputs(stderr, identify_value(val1));Outputs 1 to the terminal window
api_dlg_get_resources(editH, API_RN_VALUE, &val1);
fputs(stderr, identify_value(val1));Outputs “R” to the terminal window
```

api_dlg_set_resources

This function sets the value of a specified dialog object resource. The function does not return a value.

Syntax

```
api_dlg_set_resources(itemH, resource, value);
```

Where

itemH is the dialog object handle from which to retrieve data

resource is an enumerated integer value that designates the resource value to retrieve

value is the value to assign to the specified resource of the object

Examples

```
api_dlg_set_resources(checkH, API_RN_TOGGLE_STATE, TRUE);
api_dlg_set_resources(editH, API_RN_VALUE, "L");
```

Layered API Dialog Elements

The following descriptions are the enumerated item types that can be specified in the api_dlg_create_item function to create a new dialog object. Each description includes the callbacks that can be assigned to the dialog object and the resources that are available for the dialog box object.

API_CHECK_BUTTON_ITEM

This is a toggle button item. Each time the user clicks on the object, it will toggle between *on* or *off*.

Callbacks

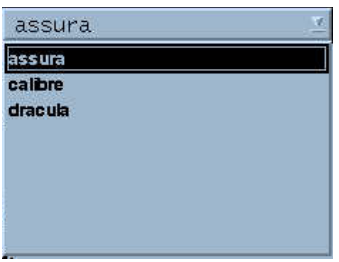
Called when the user toggles the button:

```
API_VALUE_CHANGED_CALLBACK
```

Resources

Resource	Type	Default	Description
API_RN_CAPTION	string	“”	caption
API_RN_MANAGE_FLAG	boolean	TRUE	item is displayed
API_RN_SENSITIVE_FLAG	boolean	TRUE	item is sensitive
API_RN_TOGGLE_STATE	boolean	FALSE	on or off state
API_RN_USE_BITMAP	boolean	FALSE	use bitmap caption
API_RN_UP_BUTTON_BITMAP	string	“”	off bitmap name
API_RN_DOWN_BUTTON_BITMAP	string	“”	on bitmap name

API_DROPDOWNLIST_COMBO_ITEM



This dialog box object consists of a single text field which has a pop-up list of available elements that can be chosen as the value. The text field displays the currently selected value.

Callbacks

Called when an item is selected by clicking on it in the pop-up dialog box area:

`API_LIST_SELECTION_CALLBACK`

Called when an item is double clicked in the pop-up dialog box area:

`API_LIST_DBLCLK_CALLBACK`

Resources

Name	Type	Default	Description
API_RN_CAPTION	string	“”	label
API_RN_MANAGE_FLAG	boolean	TRUE	item is displayed

Name	Type	Default	Description
API_RN_SENSITIVE_FLAG	boolean	TRUE	item is sensitive
API_RN_VISIBLE_ITEM_COUNT	int	8	Number of Visible items in popup window
API_RN_ITEM_COUNT	int	0	Number of items in the list
API_RN_ITEMS	list	NULL	list of string items
API_RN_SELECTED_INDEXES	list	list(0)	For a multi-selection list, the list of selected indexes
API_RN_SELECTED_INDEX	int	0	Selected item number

API_EDIT_TEXT_ITEM

This is a text window for text editing. A caption is displayed above the text editing area. You can set up the text window to be a single or multi lined editing area.

Callbacks

Called when the edit text item gets the focus:

`API_FOCUS_CALLBACK`

Called when the edit text item loses the focus:

`API_LOSING_FOCUS_CALLBACK`

Called when the text of the edit text item is changed:

`API_VALUE_CHANGED_CALLBACK`

Called when the RETURN key is pressed if the edit text item is a single line edit text item:

`API_ACTIVATE_CALLBACK`

Arguments

Name	Type	Default	Description
API_RN_CAPTION	string	“”	Label
API_RN_MANAGE_FLAG	boolean	TRUE	item is displayed
API_RN_SENSITIVE_FLAG	boolean	TRUE	item is sensitive
API_RN_EDITABLE	boolean	TRUE	item is editable
API_RN_VALUE	string	“”	text value
API_RN_EDIT_MODE	int	API_RV_SINGLE_LINE_EDIT	API_RV_SINGLE_LINE_EDIT or API_RV_MULTI_LINE_EDIT
API_RN_COLUMNS	int	20	number of columns
API_RN_ROWS	int	8	number of rows for multi-edit item
API_RN_USE_SCROLL_BARS	boolean	TRUE	display a scroll bar on multi-line edit items.

API_LABEL_ITEM

Comments:

A text label or a bitmap image.

Callbacks

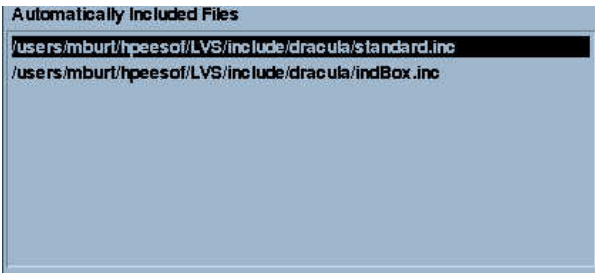
None

Resources

Name	Type	Default	Description
API_RN_CAPTION	string	“”	label text
API_RN_MANAGE_FLAG	boolean	TRUE	item is displayed
API_RN_SENSITIVE_FLAG	boolean	TRUE	item is sensitive
API_RN_USE_BITMAP	boolean	FALSE	use a bitmap instead of text

Name	Type	Default	Description
API_RN_UP_BUTTON_BITMAP	string	“”	Name of the bitmap
API_RN_DOWN_BUTTON_BITMAP	string	“”	Name of the bitmap to use if the item is not sensitive

API_LIST_ITEM



A scrollable list of strings.

Callbacks

Called when the user makes a selection change in the list:

`API_LIST_SELECTION_CALLBACK`

Called when user double clicks on an item on the list:

`API_LIST_DBLCLK_CALLBACK`

Called when user scrolls the list so that a new item becomes the first index being shown:

`API_LIST_TOP_INDEX_CHANGED_CALLBACK`

Resources

Name	Type	Default	Description
API_RN_CAPTION	string	“”	label
API_RN_SELECTION_POLICY	int	single	Single or multiple selections, API_RV_SINGLE_SELECTION or API_RV_MULTIPLE_SELECTION.
API_RN_MANAGE_FLAG	boolean	TRUE	item is displayed
API_RN_SENSITIVE_FLAG	boolean	TRUE	item is sensitive
API_RN_VISIBLE_ITEM_COUNT	int	8	number of visible items
API_RN_ITEM_COUNT	int	0	number of items
API_RN_ITEMS	list	NULL	list of strings
API_RN_SELECTED_INDEXES	list	list(0)	for multi-select list, list of the selected items
API_RN_SELECTED_INDEX	int	0	selected item
API_RN_SELECTED_ITEM_COUNT	int	0	Number of selected items
API_RN_SELECTED_ITEMS	list	NULL	string list of selected items
API_RN_VISIBLE_COLUMN_COUNT	int	6	number of visible columns
API_RN_TOP_INDEX	int	0	Position of the item that is the first visible item in the list
API_RN_SCROLL_BAR_CONFIG	int	API_RV_LIST_BOTH_SCRBAR	specify if vertical or horizontal scroll bar is needed (don’t change this)

API_PUSH_BUTTON_ITEM

This button can use a text label or a bitmap image. The item is activated when the user left clicks it.

Callbacks

Called when the button item is activated with the mouse:

`API_ACTIVATE_CALLBACK`

Called when the left mouse button is down, and the cursor is over the button:

`API_LEFT_BUTTON_DOWN_CALLBACK`

Called when the left mouse button is released, and the cursor is over the button:

`API_LEFT_BUTTON_UP_CALLBACK`

Resources

Name	Type	Default	Description
API_RN_CAPTION	string	“”	button text label
API_RN_MANAGE_FLAG	boolean	TRUE	item is displayed
API_RN_SENSITIVE_FLAG	boolean	TRUE	item is sensitive
API_RN_USE_BITMAP	boolean	FALSE	use a bitmap
API_RN_UP_BUTTON_BITMAP	string	“”	regular bitmap
API_RN_DOWN_BUTTON_BITMAP	string	“”	button down bitmap
API_RN_FOCUSED_BUTTON_BITMAP	string	“”	Button has focus bitmap
API_RN_DISABLED_BUTTON_BITMAP	string	“”	Desensitized button bitmap

API_TABLE_GROUP

The table item is used to design dialog boxes. The table item is a container that controls the size and location of the dialog box objects. The item uses an array model to simplify the arrangement of child objects. The table item also controls formatting for dialog boxes.

Items inside the table are placed at row and column locations in a variable size array. Items may span more than one row or column. The array can expand or contract in size as needed. There are options to control justification and place size restrictions on rows and columns of items.

The table can contain any number and type of children, including other table items.

Callbacks

None

Resources

Name	Type	Default	Description
API_RN_CAPTION	string	“”	Title for the table
API_RN_MANAGE_FLAG	boolean	TRUE	table is displayed
API_RN_SENSITIVE_FLAG	boolean	TRUE	table is enabled
API_RN_ROW_SPACING	int	2	pixels between rows
API_RN_COLUMN_SPACING	int	2	pixels between columns
API_RN_MARGIN_HEIGHT	int	2	top and bottom margins in pixels
API_RN_MARGIN_WIDTH	int	2	left and right margins in pixels
API_RN_FRAME_VISIBLE	boolean	FALSE	frame visible
API_RN_DEFAULT_OPTIONS	int	0	Sets default table options for children
API_RN_NUM_COLORS	int	1	number of rows or columns, depending on orientation
API_RN_ORIENTATION	int	API_RV_VERTICAL	API_RV_VERTIVAL or API_RV_HORIZONTAL.
API_RN_EQUALIZE_ROW	int	FALSE	each item in the row has the same size
API_RN_EQUALIZE_COLUMN	int	FALSE	each item in the column has the same size

Name	Type	Default	Description
API_RN_EQUALIZE_ALL	int	FALSE	All items have the same size
API_RN_RADIO_BEHAVIOR	boolean	TRUE	If TRUE, all radio buttons will have the one of many behavior

All child items in the table container will have the following additional resources:

Name	Type	Default	Description
API_RN_TBL_ROW_POSITION	int	API_RV_TBL_DEF_ROW	row position of the child item
API_RN_TBL_COL_POSITION	int	API_RV_TBL_DEF_COL	column position of the child item
API_RN_TBL_OPTIONS	int	API_RV_TBL_DEF_OPT	see "Layered API Table Options" on page B-12
API_RN_TBL_ROW_SPAN	int	1	number of rows the child item spans
API_RN_TBL_COL_SPAN	int	1	number of columns the child item spans

Layered API Table Options

You can or the following enumerations produce justifications and size adjustments for tables and child objects of tables:

- API_RV_TBL_LEFT

  Horizontally left justified

- API_RV_TBL_RIGHT

  Horizontally right justified

- API_RV_TBL_TOP

  Vertically top justified

- API_RV_TBL_BOTTOM

  Vertically bottom justified
- API_RV_TBL_LK_WIDTH

  Do not expand the item horizontally
- API_RV_TBL_LK_HEIGHT

  Do not expand the item vertically
- API_RV_TBL_SM_HEIGHT

  Force the column to be the minimum height
- API_RV_TBL_SM_WIDTH

  Force the column to be the minimum width
- API_RV_FIX_WIDTH

  Equivalent to setting: API_RV_TBL_LK_WIDTH|API_RV_TBL_SM_WIDTH
- API_RV_FIX_HEIGHT

  Equivalent to setting:
  API_RV_TBL_LK_HEIGHT|API_RV_TBL_SM_HEIGHT
- API_RV_FIX_SIZE

  Equivalent to setting: API_RV_FIX_WIDTH|API_RV_FIX_HEIGHT

Index