



Agilent Technologies

Advanced Design System 2011.01

**February 2011
Netlist Exporter Setup**

© **Agilent Technologies, Inc. 2000-2011**

5301 Stevens Creek Blvd., Santa Clara, CA 95052 USA

No part of this documentation may be reproduced in any form or by any means (including electronic storage and retrieval or translation into a foreign language) without prior agreement and written consent from Agilent Technologies, Inc. as governed by United States and international copyright laws.

Acknowledgments

Mentor Graphics is a trademark of Mentor Graphics Corporation in the U.S. and other countries. Mentor products and processes are registered trademarks of Mentor Graphics Corporation. * Calibre is a trademark of Mentor Graphics Corporation in the US and other countries. "Microsoft®, Windows®, MS Windows®, Windows NT®, Windows 2000® and Windows Internet Explorer® are U.S. registered trademarks of Microsoft Corporation. Pentium® is a U.S. registered trademark of Intel Corporation. PostScript® and Acrobat® are trademarks of Adobe Systems Incorporated. UNIX® is a registered trademark of the Open Group. Oracle and Java and registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. SystemC® is a registered trademark of Open SystemC Initiative, Inc. in the United States and other countries and is used with permission. MATLAB® is a U.S. registered trademark of The Math Works, Inc.. HiSIM2 source code, and all copyrights, trade secrets or other intellectual property rights in and to the source code in its entirety, is owned by Hiroshima University and STARC. FLEXIm is a trademark of Globetrotter Software, Incorporated. Layout Boolean Engine by Klaas Holwerda, v1.7 <http://www.xs4all.nl/~kholwerd/bool.html> . FreeType Project, Copyright (c) 1996-1999 by David Turner, Robert Wilhelm, and Werner Lemberg. QuestAgent search engine (c) 2000-2002, JObjects. Motif is a trademark of the Open Software Foundation. Netscape is a trademark of Netscape Communications Corporation. Netscape Portable Runtime (NSPR), Copyright (c) 1998-2003 The Mozilla Organization. A copy of the Mozilla Public License is at <http://www.mozilla.org/MPL/> . FFTW, The Fastest Fourier Transform in the West, Copyright (c) 1997-1999 Massachusetts Institute of Technology. All rights reserved.

The following third-party libraries are used by the NlogN Momentum solver:

"This program includes Metis 4.0, Copyright © 1998, Regents of the University of Minnesota", <http://www.cs.umn.edu/~metis> , METIS was written by George Karypis (karypis@cs.umn.edu).

Intel@ Math Kernel Library, <http://www.intel.com/software/products/mkl>

SuperLU_MT version 2.0 - Copyright © 2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from U.S. Dept. of Energy). All rights reserved. SuperLU Disclaimer: THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS

INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

7-zip - 7-Zip Copyright: Copyright (C) 1999-2009 Igor Pavlov. Licenses for files are: 7z.dll: GNU LGPL + unRAR restriction, All other files: GNU LGPL. 7-zip License: This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA. unRAR copyright: The decompression engine for RAR archives was developed using source code of unRAR program. All copyrights to original unRAR code are owned by Alexander Roshal. unRAR License: The unRAR sources cannot be used to re-create the RAR compression algorithm, which is proprietary. Distribution of modified unRAR sources in separate form or as a part of other software is permitted, provided that it is clearly stated in the documentation and source comments that the code may not be used to develop a RAR (WinRAR) compatible archiver. 7-zip Availability: <http://www.7-zip.org/>

AMD Version 2.2 - AMD Notice: The AMD code was modified. Used by permission. AMD copyright: AMD Version 2.2, Copyright © 2007 by Timothy A. Davis, Patrick R. Amestoy, and Iain S. Duff. All Rights Reserved. AMD License: Your use or distribution of AMD or any modified version of AMD implies that you agree to this License. This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. AMD Availability: <http://www.cise.ufl.edu/research/sparse/amd>

UMFPACK 5.0.2 - UMFPACK Notice: The UMFPACK code was modified. Used by permission. UMFPACK Copyright: UMFPACK Copyright © 1995-2006 by Timothy A. Davis. All Rights Reserved. UMFPACK License: Your use or distribution of UMFPACK or any modified version of UMFPACK implies that you agree to this License. This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at

your option) any later version. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. UMFPACK Availability: <http://www.cise.ufl.edu/research/sparse/umfpack> UMFPACK (including versions 2.2.1 and earlier, in FORTRAN) is available at <http://www.cise.ufl.edu/research/sparse> . MA38 is available in the Harwell Subroutine Library. This version of UMFPACK includes a modified form of COLAMD Version 2.0, originally released on Jan. 31, 2000, also available at <http://www.cise.ufl.edu/research/sparse> . COLAMD V2.0 is also incorporated as a built-in function in MATLAB version 6.1, by The MathWorks, Inc. <http://www.mathworks.com> . COLAMD V1.0 appears as a column-preordering in SuperLU (SuperLU is available at <http://www.netlib.org>). UMFPACK v4.0 is a built-in routine in MATLAB 6.5. UMFPACK v4.3 is a built-in routine in MATLAB 7.1.

Qt Version 4.6.3 - Qt Notice: The Qt code was modified. Used by permission. Qt copyright: Qt Version 4.6.3, Copyright (c) 2010 by Nokia Corporation. All Rights Reserved. Qt License: Your use or distribution of Qt or any modified version of Qt implies that you agree to this License. This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301USA Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. Qt Availability: <http://www.qtsoftware.com/downloads> Patches Applied to Qt can be found in the installation at: `$HPEESOF_DIR/prod/licenses/thirdparty/qt/patches`. You may also contact Brian Buchanan at Agilent Inc. at brian_buchanan@agilent.com for more information.

The HiSIM_HV source code, and all copyrights, trade secrets or other intellectual property rights in and to the source code, is owned by Hiroshima University and/or STARC.

Errata The ADS product may contain references to "HP" or "HPEESOF" such as in file names and directory names. The business entity formerly known as "HP EEsof" is now part of Agilent Technologies and is known as "Agilent EEsof". To avoid broken functionality and to maintain backward compatibility for our customers, we did not change all the names and labels that contain "HP" or "HPEESOF" references.

Warranty The material contained in this document is provided "as is", and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied, with regard to this documentation and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Agilent and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license. Portions of this product include the SystemC software licensed under Open Source terms, which are available for download at <http://systemc.org/> . This software is redistributed by Agilent. The Contributors of the SystemC software provide this software "as is" and offer no warranty of any kind, express or implied, including without limitation warranties or conditions or title and non-infringement, and implied warranties or conditions merchantability and fitness for a particular purpose. Contributors shall not be liable for any damages of any kind including without limitation direct, indirect, special, incidental and consequential damages, such as lost profits. Any provisions that differ from this disclaimer are offered by Agilent only.

Restricted Rights Legend U.S. Government Restricted Rights. Software and technical data rights granted to the federal government include only those rights customarily provided to end user customers. Agilent provides this customary commercial license in Software and technical data pursuant to FAR 12.211 (Technical Data) and 12.212 (Computer Software) and, for the Department of Defense, DFARS 252.227-7015 (Technical Data - Commercial Items) and DFARS 227.7202-3 (Rights in Commercial Computer Software or Computer Software Documentation).

Component Definitions	7
Component Definition Files	7
Component Definition File Setup with the GUI	11
Configuration Files	14
Configuration Files Used with Front End Flow	14
Configuration File Locations	14
Configuration File Descriptions	15
Tool Configuration Files	18
Customizing a Netlister	23
Setting Up Automatically Included Files	23
Adding Value Mapping Functions	24
Adding New Netlist Exporting Functions	28
Overriding Existing Front End Flow API Functions	30
Front End Flow Functions	33
Instance Netlist Exporting Functions	33
Subcircuit Header Functions	36
Subcircuit Footer Functions	37
Netlist Header Function	38
Netlist Footer Function	39
Circuit Output Functions	39
Parameter Formatting Functions	40
Global Variable Functions	41
Core Functions	43
Hspice Netlister Example	52
Creating the New Dialect Directories and Files	52
Modifying the Configuration File as Needed	53
Modifying the Netlisting Functions as Needed	54
Creating Component Definitions	60
Verifying the Netlist	70
Setting up GUI Options	72
Option List Global Variable	72
Setup	75
License Requirements	75
Installing Netlist Exporter	75
Configuration File Settings	76
Design Tool Support	78
Front End Flow Directory Structure	79
Adding Tools to Front End Flow	79

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* (felowlc) 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 (see *Instance Netlist Exporting Functions* (felowlc)).
- 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 output 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* (felowlc) for information. Check the documentation for the tool you are using to verify the value outputs that it needs.

Component Definition File Editing

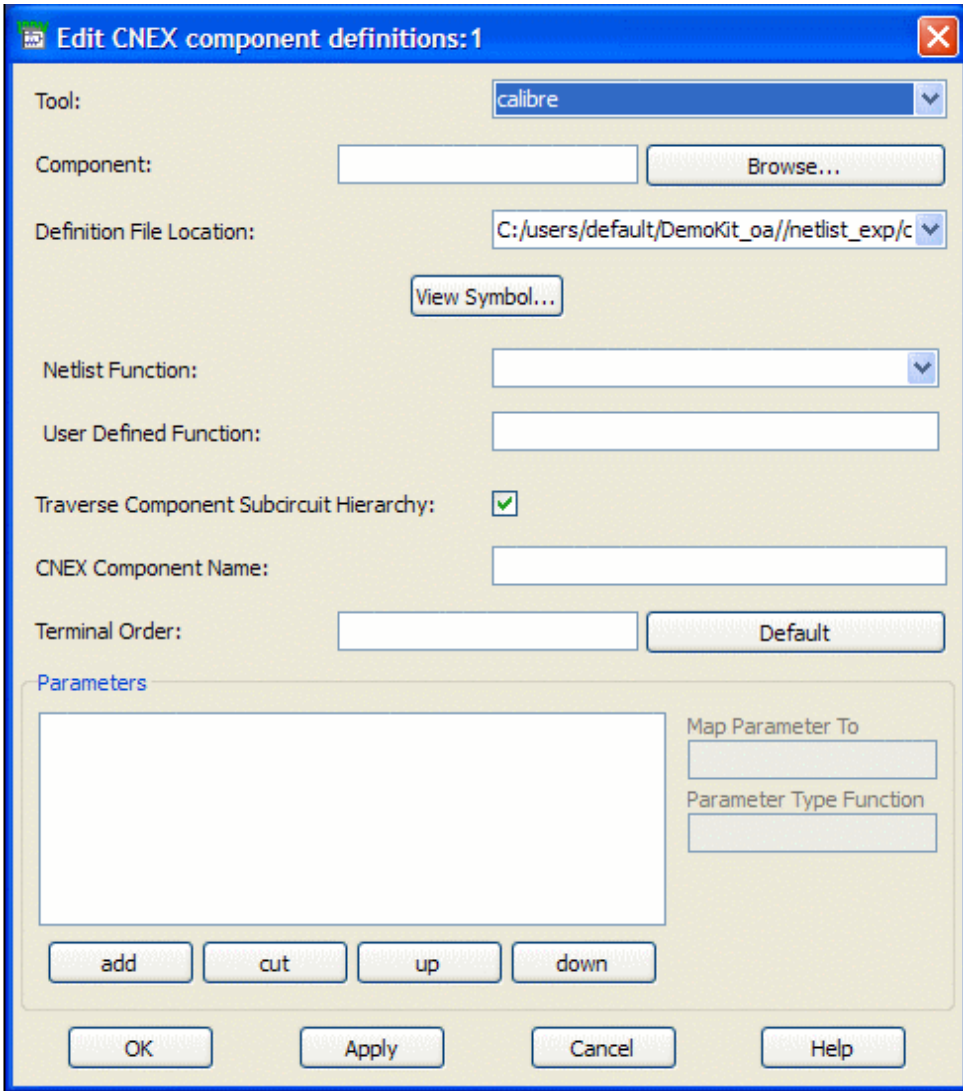
There are the following two ways to edit component definition files:

- 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](#).

Component Definition File Setup with the GUI

The *Component Definition Editor* ([Component Definition Editor](#)) 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.



Component Definition Editor

Tool:	Choose the tool you want from drop-down menu. The default tool when you start the <i>Component Definition Editor</i> 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 Window. 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 <i>Component Dialog Editor</i> will follow the component definition path contained in the <i>CNEX_COMPONENT_PATH</i> (felowlc) 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. <i>Instance Netlist Exporting Functions</i> (felowlc) 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 <i>cnexSubcircuitInstance</i> if the component is a subcircuit, or <i>cnexUnknownInstance</i> if the component is a primitive.
CNEX Component Name:	Type in the name you want for the component instance. If use the <i>cnexSubcircuit</i> netlist function (see Netlist Function: above), you can use <i>subcircuit</i> 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.
Parameters:	This list box contains the parameters you want to output. The parameters are output in the order listed.
Map Parameter To	This field displays the netlist name for the parameter selected in the <i>Parameters</i> list box.
Parameter Type Function	This field displays the AEL mapping function for the parameter selected in the <i>Parameters</i> list box.
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.

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.

Configuration Files

There are several text configuration files that define variables for the default Front End Flow netlister and variables for specific tools. This section 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:

<i>de_sim.cfg</i>	The <i>de_sim.cfg</i> file is the PDE configuration file. This file is used to load Front End Flow.
<i>CNEX.cfg</i>	The <i>CNEX.cfg</i> file is the Front End Flow configuration file. This file contains information used by the core functions of the Front End Flow netlister.
<i><tool>.cfg</i>	The <i><tool>.cfg</i> configuration file contains specific data for a tool. The configuration data consists of options that are output into netlists for a particular tool.
<i>CNEX_config.<tool></i>	The <i>CNEX_config.<tool></i> 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 workspace 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 workspace 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 workspace directory
- \$HOME/hpeesof/config
- \$HPEESOF_DIR/custom/config
- \$HPEESOF_DIR/config

Note
Do not modify variable values within the workspace directory *CNEX.cfg* file. The workspace 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 workspace directory *CNEX.cfg* file. The workspace 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:

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 workspace 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 workspace 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 workspace 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 workspace 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 workspace 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 in a given vocabulary. 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

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 character 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 resistor 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_CONTINUATION_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_CONTINUATION_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* for Assura and Dracula and *N\$* for Calibre.



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_SCIENTIFIC* function maps scalar quantities into scientific notation. The *SCALAR_TO_SCIENTIFIC* function is useful if your tool's netlist format does not support scalars. See *SCALAR_UNIT_MAPPING* for information on customizing scalar mapping. The *SCALAR_TO_SCIENTIFIC* default value is *FALSE* . (No function line present equals *FALSE* .)

For example, if *SCALAR_TO_SCIENTIFIC* 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.

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 *Front End Flow Functions* (felowlc). 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 *Front End Flow Functions* (felowlc).

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 *Netlist Exporter* (felowug) 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 DesignContext 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.

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.

Accessing Parameter Values Other Than the Current Parameter Value

In certain cases, it is necessary to access parameter values other than the current parameter value. This can be used to create an expression of multiple parameter values. An example of this would be a component that has Width and Length parameters, but the output netlist requires that an area parameter be output which is equal to the Width times the Length. The function **cnexGetParamValueByName()** was added in ADS 2003A. Using this function, the name of the desired parameter is passed in, and the value is returned.

Example

```
defun myFoundryArea (value)
{
decl paramIter = db_create_param_iter(cnexCurrentInst);
if(db_param_iter_is_valid(paramIter))
{
decl Width=cnexGetParamValueByName("Width",paramIter);
decl Length=cnexGetParamValueByName("Length",paramIter);
```

```

if(Width && Length)
{
    return(strcat(Width, " * ", Length));
}
}
return("");
}

```

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* (feflowlc) 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 in a given vocabulary. All files named *cnexGlobals* or *cnexNetlistFunctions* in any path contained in the *CNEX_EXPORT_FILE_PATH* (feflowlc) use a single common namespace. Therefore any AEL function in these files has access to all other AEL variables and functions in such files.

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 *Configuration Files* (feflowlc) for more information on *CNEX_EXPORT_FILE_PATH* (feflowlc).

Example 1: Placing the Function

You want to place the function *myFoundryAddQuotes* so that Front End Flow can use it.

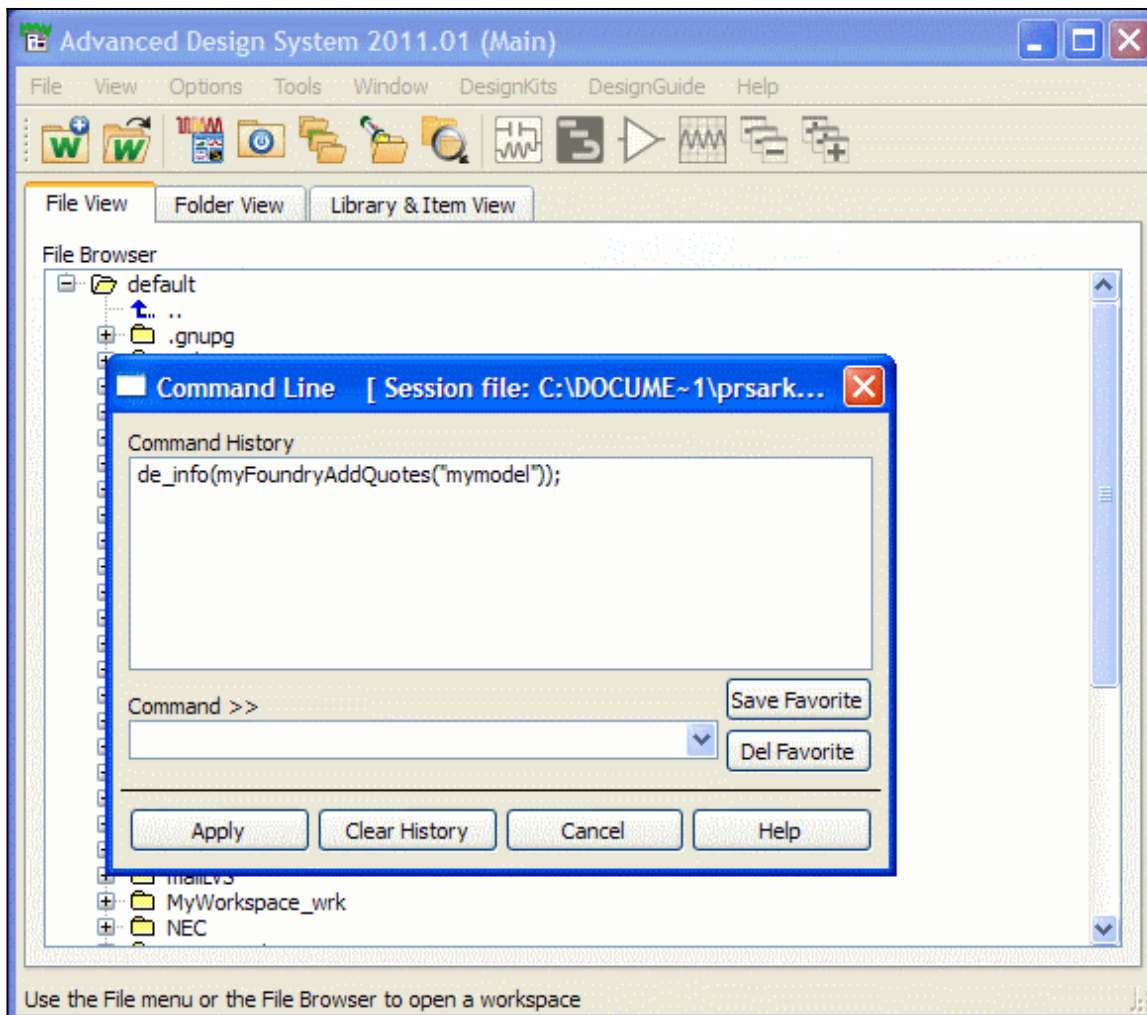
Add a file named *cnexNetlistFunctions* to the directory `{%CNEX_INSTALL_DIR}/ael/{%CNEX_TOOL}` . If you wrote the function for HSpice, the directory would be `$HPEESOF_DIR/netlist_exp/ael/HSpice` . The next time that a Front End Flow netlist is generated, the *myFoundryAddQuotes* function will be available.

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 *de_info* command and typing the function into the *Command Line* . This is shown in [Accessing the AEL Command Line to Verify Functions](#).



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 *de_info* dialog .



Note

The *de_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 *Hspice Netlister Example* (felowlc).

Placing a New Netlist Exporting Function

The configuration variable *CNEX_EXPORT_FILE_PATH* (felowlc) 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* (felowlc) 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 *Configuration Files* (felowlc) for more information on the *CNEX_EXPORT_FILE_PATH* (felowlc).

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. *Front End Flow Functions* (felowlc) 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 *Front End Flow Functions* (felowlc).

- 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* (felowlc). The values it receives are the design name and the design context 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, context)
{
    return(cnexOutputSubcircuitHeader(designName, context));
}
```

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, context)
{
    return("");
}
```

Place this function into a file called *cnexNetlistFunctions* . Put this file in an HSpice subdirectory in one of the *CNEX_EXPORT_FILE_PATH* (felowlc) 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 *Hspice Netlister Example* (felowlc).

Function De

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, context)
{
decl net=originalCnexOutputTopcellHeader(designName, context);
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* .

Front End Flow Functions

This section contains a list of the default Front End Flow functions. Every entry has a description of the function, its arguments, and its return value. You can override any of these functions unless specifically noted. However, 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 , context );
```

Where

designName is a string containing the design name

context is the DesignContext handle of the design

Example

```
decl context = de_get_design_context_from_name("mylib:mydesign:schematic");
decl net=cnexOutputSubcircuitHeader("mylib:mydesign:schematic", context);
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 , context );
```

Where

designName is a string containing the top cell design name

context is the DesignContext 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 , context );
```

Where

designName is a string containing the top cell design name
context is the DesignContext handle to the top cell

Example

```
decl context = de_get_design_context_from_name("mylib:mydesign:schematic");
decl net=cnexOutputSubcircuitFooter("mylib:mydesign:schematic", context);
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 , context );
```

Where

designName is a string containing the top cell design name
context is the DesignContext 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 , context );
```

Where

designName is a string containing the top cell design name

context is the DesignContext 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 , context );
```

Where

designName is a string containing the top cell design name

context is the DesignContext 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

```
cnexOutputCircuitData( context, topLevel );
```

Where

context is a DesignContext 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("1m0hm");
```

returns *1e-3*

cnexGetParamValueByName

This function will retrieve the value of the specified parameter for the instance that is currently being netlisted. If the parameter name specified on the instance is not found, the function will return NULL.

Syntax

```
cnexGetParamValueByName( paramName );
```

Where

paramName is the string value of a parameter on the instance that is currently being netlisted.

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

```
cnexExportClearGlobals();
```

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 *Setting up GUI Options* (feflowlc).

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.

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( context );
```

Where

context is the DesignContext handle of the schematic design 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

cnexExportShortName

This function will return the leaf name of a full path name that is passed in. This function is normally used to get the component name from a full directory path.

Syntax

```
cnexExportShortName( name );
```

Where

name is the string value of a component name that is expressed with a full path.

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( context );
```

Where

context is the DesignContext of the schematic design 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.

Hspice Netlister Example

This section 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. [Create the New Dialect Directories and Files.](#)
2. [Modify the Configuration File as Needed.](#)
3. [Modify the Netlisting Functions as Needed.](#)
4. [Create Component Definitions.](#)
5. [Verify the Netlist.](#)

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* (feflowlc), Front End Flow checks the variable `CNEX_COMPONENT_PATH` (feflowlc) 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` (felowlc) 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* (felowlc). 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 *Configuration Files* (felowlc) 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](#), 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](#)
2. [Modifying Header and Footer Functions](#)

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 workspace so that no pre-existing information is used. For the example in this section a new workspace, `HSpiceSetup_awk`, is created. Next, a new schematic 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 then 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 **Window > 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 documentation, 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 for *test1*.
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 documentation, 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 *cnexGlobalNodeInstance* 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 documentation 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 documentation 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 documentation 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, context)
{
    return(cnexOutputSubcircuitHeader(designName, context));
}
```

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, context)
{
    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, context)
{
    return(cnexOutputSubcircuitFooter(designName, context));
}
```

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, context)
{
    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 section deals with components that are delivered with ADS. Your foundry kit components should fall into these categories. Because ADS has hundreds of components. This section 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 <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>
```

or a polynomial form:

```
Cxxx n1 n2 POLY c0 c1...
```

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 *Component Definitions* (feflowlc), 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 section 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 section, 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 documentation 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 *Temp* 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* (feflowlc), 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 documentation , the following is the format for the BJT:

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

or

```
Qxxx nc nb ne mname <AREA = area> <AREAB = val>
+ <AREAC = val> <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 documentation.

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 section 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* (feflowlc), 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 *DevOpPtLevel*.

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 is 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* . One 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.

Setting up GUI Options

Netlist Exporter (felowug) 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* (felowlc). 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

Advanced Design System 2011.01 - Netlist Exporter Setup
configuration file. That way, you can set the global options list up by calling the function
anywhere within your own code.

Setup

This section covers the installation configuration file settings for Front End Flow. For information on using Front End Flow, refer to the *Netlist Exporter* (felowug).

License Requirements

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

- *trans_spice_netlist*

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 *Windows License Setup* (instalpc) or *ADS License Setup - Unix and Linux* (install).

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 *UNIX and Linux Installation* (install) to run the SETUP utility and load the *install* program.
For a PC installation, follow the instructions in *Windows Installation* (instalpc). 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:
 - **Complete** - If you choose a *Complete* installation, the Netlist Exporter will be automatically installed.
 - **Custom** - If you choose a *Custom* installation, the Netlist Exporter will be automatically.
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 *UNIX and Linux Installation* (install) or *Windows Installation* (instalpc).

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 inside your workspace. 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 *Component Definitions* (feflowlc) 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

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](#) 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:

<i>ael</i>	The <i>ael</i> directory contains the compiled AEL (atf) program files with or without the associated AEL files. Each directory specified in <i>CNEX_EXPORT_FILE_PATH</i> will be searched for AEL files relevant to Front End Flow (see <i>CNEX_EXPORT_FILE_PATH</i> (feflowlc)). The relevant files will be loaded. For more information on AEL files as they relate to Front End Flow, see <i>Customizing a Netlister</i> (feflowlc), and <i>Setting up GUI Options</i> (feflowlc).
<i>components</i>	The <i>components</i> 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 <i>Component Definitions</i> (feflowlc) for information on component definition files.
<i>config</i>	The <i>config</i> directory can be found in the following three locations:
<i>include</i>	The files to automatically include in a netlist are specified within the tool-specific subdirectory of the <i>include</i> directory. See <i>Customizing a Netlister</i> (feflowlc), 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 *Creating Netlists* (feflowug). 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 *Configuration Files* (feflowlc) 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 *Customizing a Netlister* (feflowlc) 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.