

NOTICE: This document contains references to Agilent Technologies. Agilent's former Test and Measurement business has become Keysight Technologies. For more information, go to **www.keysight.com**.





September 2011
User-Defined Models

© 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 02110-1301 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. 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).

Building User-Compiled Analog Models	8
About User-Compiled Model Code	18
Creating Linear Circuit Elements	38
Creating Nonlinear Circuit Elements	52
Creating Transient Circuit Elements	62
Custom Modeling with Symbolically-Defined Devices	67
Custom Modeling with Frequency-Domain Defined Devices	102
Building Signal Processing Models	125
Writing Component Models	139
Data Types for Model Builders	184
Porting UC Berkeley Ptolemy Models	221
User-Defined Models API Reference	223
active_noise	231
add_lin_n	232
add_lin_y	233
add_nl_gc	234
add_nl_iq	235
add_tr_capacitor	236
add_tr_gc	237
add_tr_inductor	238
add_tr_iq	239
add_tr_lossy_inductor	240
add_tr_mutual_inductor	241
add_tr_resistor	242
add_tr_tline	243
dump_params	244
ee_compute_n	245
ee_compute_y	246
ee_post_analysis	247
ee_pre_analysis	248
first_frequency	249
first_iteration	250
get_ucm_num_external_nodes	251
get_ucm_num_of_params	252
get_ucm_param_complex_value	253
get_ucm_param_data_type	254
get_ucm_param_int_value	255
get_ucm_param_name	256
get_ucm_param_num_repeats	257
get_ucm_param_ptr	258
get_ucm_param_real_value	259
get_ucm_param_string_value	260
get_ucm_param_vector_complex_value	261
get_ucm_param_vector_int_value	262
get_ucm_param_vector_real_value	263
get_ucm_param_vector_size	264
get_delay_v	265
get_params	266
get_temperature	267
get_tr_time	268
get_user_inst	269

User-Defined Models

is_ucm_repeat_param	270
load_elements	271
load_elements2	272
multifile	273
passive_noise	274
print_ucm_param_value	275
send_error_to_scn	276
send_info_to_file	277
send_info_to_scn	278
s_y_convert	279
verify_senior_parameter	280

Building User-Compiled Analog Models



Note

User-Compiled Models cannot be created using Schematic view.

User-defined element models are implemented in ANSI-C code. The user-written code is then compiled and linked with supplied object code to make a dynamically loaded shared library. While the equation and parametric circuit capabilities included in the circuit simulators can be used to effectively alter an element or network response through its parameters, the user-compiled model feature allows you access to state vector voltages that affect the model's response currents and charges.

The analysis code for user-compiled models can be written to influence its response depending on its parameters, stimulus controls, analysis type, and pin voltages. The user-defined code can make use of many built-in element models.

Creating a model consists of four main steps:

- Defining the parameters whose values will be entered from the schematic. See, [Defining Model Parameters](#)
- Defining the symbol and the number of pins. See, [Working with Symbols](#)
- Writing the C or C++ code itself
- Compiling. See, [Compiling the Model](#)

When appropriately coded, these elements can be used in linear, nonlinear (Harmonic Balance), transient, and Circuit Envelope simulations.

User-Compiled Model example can be found at ADS examples/Tutorial/UserCompiledModel_wrk.




Note

To use this tool, you must have the appropriate C_+ compiler installed on your computer. For details, refer to the installation documentation for your platform.

Working with Symbols

UserCompiled Model has symbol view only. A UserCompiled Model symbol can be created, viewed or edited with ADS standard symbol generation tool.

Follow the steps below to generate a symbol for a new UserCompiled Model for example *myModel*

1. From ADS main window, choose **File > New > Symbol** or click  on the tool bar to open New Symbol dialog box.
2. Specify the Cell name as *myModel*.
3. Click **OK** to open the Symbol view window.
4. Define the symbol. Refer to *Working with Symbols* (usrguide).
5. Click **Save AEL file** followed by **OK** to save the symbol.

To open the symbol for an existing UserCompiled Model for example *myModel*, choose **File**

> **Open** > **Symbol** from the ADS Main window.

Defining Model Parameters

UserCompiled Model parameters can be defined from **File** > **Design Parameters** in the model symbol view window. For more information, refer to *Create Item* (usrguide).

Model File Requirements

Schematic related files are required for both UserCompiled Model compilation and simulation. To compile a UserCompiled Model successfully, you need to have model implementation related files. To use a UserCompiled Model in a simulation, you need to have simulation related files.

Required Files

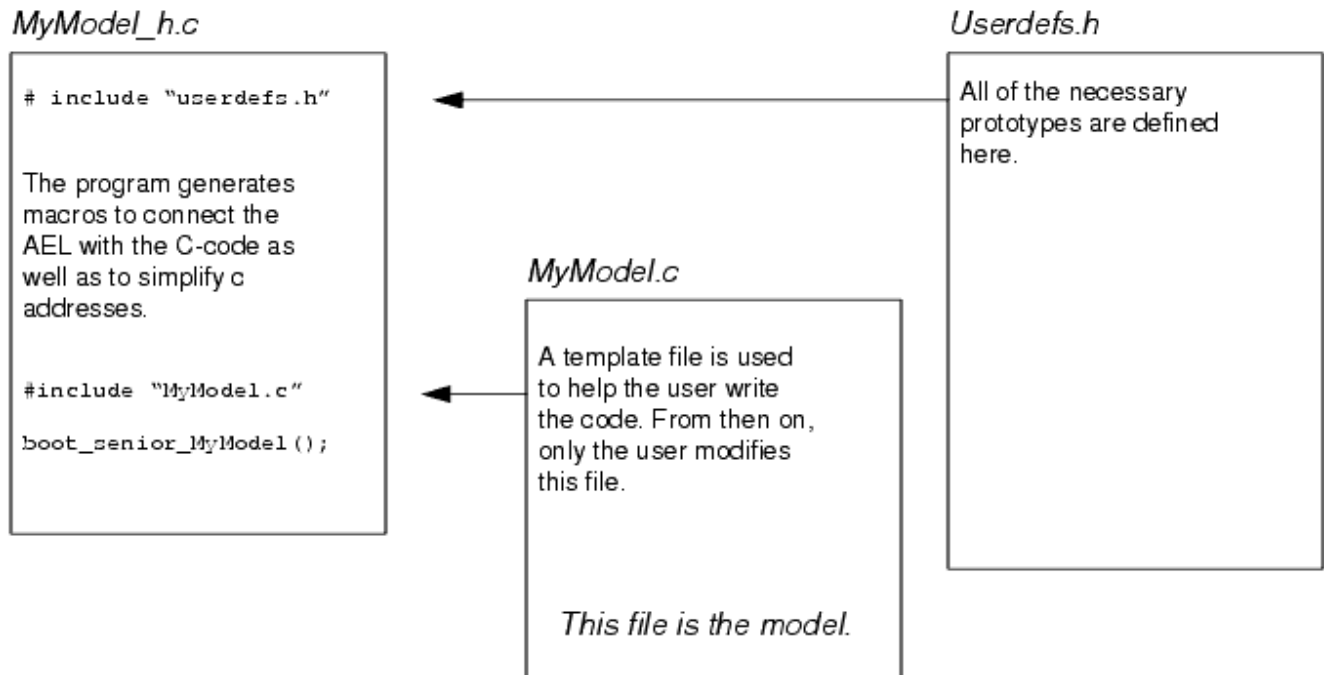
The following table lists the required files associated with a user-compiled model using the model name *MyModel* as an example.

File Name	Description
Schematic related files Schematic related files symbol definition and itemDef.ael are placed in the current_workspace/current_library/_MyModel_ sub-directory.	
itemDef.ael	Model parameter definition. Please refer to Defining Model Parameters for how to create or modify the file.
Model implementation related files These files are placed in the current workspace {{userCompiledModel/source}} directory. [†]	
MyModel.c	This is the model implementation. Model behaviors are defined in this file. The file \$HPEESOF_DIR/modelbuilder/lib/cui_circuit.template can be used as a template.
MyModel_h.c or MyModel_h.cxx	Contains the model data structure declarations. The ADS Model Builder interface generates the file based on the compile option settings when the <i>Compile</i> button is selected. MyModel_h.c is generated by ADS if C compiler is chosen in compile option, otherwise MyModel_h.cxx is generated.
MyModel_DYNAMIC.cxx	This file is used to build device database. If dynamic link is selected (.dll, .so), the ADS Model Builder interface generates the file based on the models selected. The template is available at \$HPEESOF_DIR/modelbuilder/lib.
userdefs.h	Contains ADS internal data structure declarations. This should not be modified. The template is available at \$HPEESOF_DIR/modelbuilder/lib.
makefile	This is the makefile. The ADS Model Builder interface generates the file based on the compile option settings. The template is available at \$HPEESOF_DIR/modelbuilder/lib.
user.mak	The user-customized make options should be added in this file since makefile might be overwritten by the ADS Model Builder interface when the <i>Compile</i> button is selected. The template is available at \$HPEESOF_DIR/modelbuilder/lib.
Simulation related files The dynamic linked shared libraries and device database file are placed under the current workspace's userCompiledModel/lib.\$SIMARCH.	
MyModel.so (Linux, Solaris)MyModel.dll (Windows)	The user-compiled model's dynamically-loaded shared library.
deviceidx.db	Device database file. Contains device and shared library mapping information.

C-Code File Relationship

The following figure uses `MyModel_h.c` as an example, and shows the C-code file relationship:

C-code File Relationship



Using the User-Compiled Model

The following topic describes how to use the Model Development Kit.

Opening a Model

Follow the steps below to open Model Development Kit dialog:

1. Open the model symbol view window. For information on symbol view, refer to [Working with Symbols](#).
2. From the symbol view window, choose **Tools > User Compiled Model > Open User Compiled Model** to open Model Development Kit dialog box. If the model is an existing model, the dialog window is initialized based on model information defined in `MyModel_h.c(xx)` header file. If the model is a new model, the dialog window is initialized with default values.

The following figure displays the Model Development Kit:

The following table describes the UCM Model Development parameter details:

UserCompiled Model Development Kit Parameters

Setup Dialog Name	Description
Component name	The UserCompiled Model name. It is same as symbol name and not editable.
Model	
Model type	Linear or Nonlinear. Select linear only when the model is a linear device and there in no internal node. Nonlinear should be selected if the model is nonlinear or there is internal node in the model. Linear is the default value. Define model behaviors in compute_* functions for a linear model type device. Define model behaviors in analyze_* functions for a nonlinear model type device.
No. of internal nodes	Number of internal nodes. The field is disabled if the model type is linear.
Code options	
Create new code template	Click to copy model code template to <current_workspace>/userCompiledModel/source/MyModel.c where, <i>MyModel</i> is the model name.
Edit model code	Click to open up the model source code <i>MyModel.c</i> in an editor.

User-Defined Models

Use user-defined pre-analysis function	If enabled, the simulator calls <i>pre_analysis</i> (modbuild) function during the circuit setup. The function is called only once so it is where the code for one time initialization should go. The default value is enabled.
Use user-defined post-analysis function	If enabled, the simulator calls <i>post_analysis</i> (modbuild) function right before exit. Please note <i>post_analysis</i> is executed by command-line simulations, ignored by ADS simulations. The default value is enabled.
Use user-defined modify-param function	If enabled, the simulator calls <i>modify_param</i> (modbuild) function whenever a model parameter value changes. The default value is enabled.
Use user-defined noise analysis function	If enabled, user-defined <i>analyze_ac_n</i> (modbuild) function is called in a linear simulation. The default value is enabled.
Use user-defined transient function	If enabled, user-defined <i>analyze_tr</i> (modbuild) function is called in a Transient simulation. The default value is disabled.
No. of inductors	Number of inductors in the model. It is needed only when user-defined transient function is enabled. In <i>analyze_tr</i> (modbuild) function, inductor must be added using one of the following function <i>add_tr_inductor</i> (modbuild), <i>add_tr_lossy_inductor</i> (modbuild) or <i>add_tr_mutual_inductor</i> (modbuild). The total number of calls to <i>add_tr_inductor</i> , <i>add_tr_lossy_inductor</i> and 2* <i>add_tr_mutual_inductor</i> functions must match the value given in this field.
Use user-defined fix transmission function	If enabled, user-defined <i>fix_tr</i> (modbuild) function is called in a Transient simulation. The default value is disabled.
No. of transmission lines	Number of transmission lines in the model. It is needed only when both user-defined transient function and user-defined transmission function are enabled. In <i>fix_tr</i> (modbuild) function, transmission lines must be added through function <i>add_tr_tline</i> (modbuild). Number of calls to <i>add_tr_tline</i> function must match the value given in this field.
Compile options	
Active libraries	Use the drop-down list to select the appropriate library from the list of available libraries.
UserCompiled models	List of all available UserCompiled Models under the selected library.
Add >>	Add the selected item from <i>UserCompiled Model</i> in the left side of UserCompiled models window to the right side in <i>Additional UserCompiled models to be linked</i> .
<< Remove	Removes a selected item from the list under <i>Additional UserCompiled models to be linked</i> .
Additional UserCompiled models to be linked	Additional models to be included in the dynamically loaded shared library.
Dynamic link library name	The shared library name created after a successful compilation. It is created under <i>userCompiledModel/lib.\$SIMARCH</i> subdirectory of the current workspace.
Compiler	Select the compiler used for compilation. If C compiler is selected, *.c extension is used, otherwise *.cxx extension is used for automatically generated header file.
Set debug flag on	If you check this box, the debug flag is enabled. The object code will include debug information enabling you to step through the code when the debugger is invoked. Default is checked.
Compile	Click to start compilation. Compilation status is shown in a new window. The object files are written to <i>userCompiledModel/source/obj.\$SIMARCH</i> subdirectory. The shared library and

User-Defined Models

	device index file are created in <code>userCompiledModel/lib.\$SIMARCH</code> subdirectory.
OK	Click to save all specified data in the dialog box and dismiss it. The file <i>MyModel_h.c(xx)</i> is generated based on data specified in the dialog box.
Apply	Click to save all specified data in the dialog box. The file <i>MyModel_h.c(xx)</i> is generated based on data specified in the dialog box.
Cancel	Cancels all specified data in this box and dismiss the dialog box.

Deleting a Model

To permanently delete an existing user-compiled model, open the model symbol view window. In the Symbol view, choose **Tools > User-Compiled Model > Delete User-Compiled Model**.

Releasing a Model License

To release UserCompiled Model license, open the model symbol view window. In the Symbol view, choose **Tools > User-Compiled Model > Release User-Compiled Model License**.

Compiling the Model

In order to compile your circuit model, you must first ensure that you have the correct compiler version installed. For information on the compiler version required for model development, refer to the section: Before You Begin > Check the System Requirements in your specific installation documentation:

- *UNIX and Linux Installation* (install)
- *Windows Installation* (instalpc)

You will also need to ensure that you add the path to where the correct compiler is installed to the beginning of your PATH environment variable. For example, if you are using Linux and you have installed your C compiler under `/opt/gcc/bin`, then you would set your PATH as follows:

```
export PATH=/opt/gcc/bin:$PATH
```

Compiler environment variable setup on Windows is more complicated. Please refer to Microsoft Visual Studio documentation for details. If the compilation is done through [Model Development Kit](#), the kit takes care of Visual Studio .NET compiler environment variable setup for users.

Determining \$SIMARCH

When compiling models, the resulting files are saved in directories using names associated with the platform architecture on which you are working. When ADS is started in 32-bit mode, the model can be compiled and linked only in 32-bit mode. When ADS is started in 64-bit mode, the model can be compiled and linked only in 64-bit mode. In 32-bit mode, the \$SIMARCH values are *win32*, *linux_x86*, or *sun_sparc*. For models compiled in 64-bit mode the \$SIMARCH values can be *win32_64*, *linux_x86_64*, or *sun_sparc_64*. These may

change in the future.

Compilation Options

UserCompiled Model compilation can be done from command line or through Model Development Kit.

Compile from the Model Development Kit

The Model Development Kit can significantly simplify model developers' implementation cycle. The development kit can do the following for model developers:

- Copy model source code template over. Developers only need to implement model behaviors in *compute_y* (modbuild), or *analyze_lin* (modbuild) etc. functions.
 - Generate header files and make file required for compilation.
 - Set up environment variables for compilation on Windows.
 - Compile the model to a shared library.
 - Generate device loading index file.
- Model Development Kit uses `<current_workspace>/userCompiledModel/source` as working directory. If the directory doesn't exist, the kit creates one. The directory is where the kit writes automatically generated files and from where the kit looks for the model source code. In order to locate files required for compilation, rules for file name convention and location need to be followed. For example, the model being worked on is called *MyModel*
- *MyModel_h.c* (for C compiler) or *MyModel_h.cxx* (for C++ compiler) - header file automatically generated by the kit. It should be located at `<current_workspace>/userCompiledModel/source` directory.
 - *MyModel.c* - Model source code which defines the model behavior. It should be located at `<current_workspace>/userCompiledModel/source` directory. The file is included by *MyModel_h.c(xx)* file.
 - *MyModel*_DYNAMIC.cxx* - C++ file which defined device loading functions. *MyModel** here is the shared library name which can be defined in Model Development Kit dialog. This file is automatically generated by the kit. It is located at `<current_workspace>/userCompiledModel/source` directory.

After a successful compilation, the Model Development Kit generates the following files in `<current_workspace>/userCompiledModel/lib.$SIMARCH` directory:

- *MyModel*.so*(Unix) or *MyModel*.dll*(Windows) - shared library
- *deviceidx.db* - device index database file

Building the User-Compiled Model from the Command Line

You can manually build the program from the command line. To compile a user-defined model from the command line, *\$HPEESOF_DIR* must be set and *\$HPEESOF_DIR/bin* must be in *\$PATH*.

SIMARCH must also be set before building the program from the command line. To help you set *SIMARCH*, you can source *bootscript.sh* making sure to use the correct syntax for your shell on UNIX. For example, for ksh, enter

```
*. bootscript.sh*
```

If you want to build a 32-bit binary on a 64-bit system, you must set `EESOF_64BIT` to 0 before sourcing `bootscript.sh`. For example, for `ksh`, enter

```
{*}export EESOF_64BIT=0*
*. bootscript.sh*
```

If you want to switch back and forth between building 32-bit and 64-bit binaries, it is recommended that you use two terminal windows to avoid switching between binaries within one window.

Before proceeding with the compile and link, you must make sure that all required files are available. These files are autogenerated or copied by ADS if you build the user-compiled model from ADS. If you cannot use ADS to autogenerate those files, you can copy the files manually from the `$HPEESOF_DIR/modelbuilder/lib` directory to the `./usercompiledModel/source` directory under the current local workspace. You may also need to modify some of the files first. [Files Required to Compile Model from Command Line](#) lists the required files and whether modification is needed.

Files Required to Compile Model from Command Line

File Name	Required Modification
makefile	Modification is needed. See the comments in the template makefile which is in <code>\$HPEESOF_DIR/modelbuilder/lib</code> .
user.mak	No modification is needed except to customize the compile or link options.
userdefs.h	No modification is needed.
MyModel_DYNAMIC.cxx	Required only for dynamic link user-compiled model. This file is only an example; modification is required before using.

Creating a Dynamic Link Model

To create a dynamic link model, the file `<model_name>_DYNAMIC.cxx` is required (`<model_name>` is the name of your model). If a generated version of this file is not available, a template is available for you to modify according to the requirements. The template contains comments to help you with the modification. Copy the file `MyModel_DYNAMIC.cxx` from the `$HPEESOF_DIR/modelbuilder/lib` directory to the `./usercompiledModel/source` directory in the current workspace directory. *The template must be customized since it cannot be used as it is.* See the comments in the template for instructions about customizing the file. After the required files are in place, run the compile command.

The compile command with options is:

```
hpeesofmake [debug=1]
```

The `debug=1` option tells the compiler to compile your source code with debug

information.

Cleaning Out Previous Compilations

The command to remove files created during a previous compilation is:

```
hpeesofmake clean
```

Accessing Dynamically Loaded Devices

When you create a dynamically-loaded device from ADS, everything is handled automatically. However a dynamically-loaded device, by default, is only accessible within the workspace in which it is created. The procedure that follows describes how to make a device available to other workspace or users.

1. The simulator has a default list of directories to search, when looking for dynamically-loaded devices, as follows:

```
../networks$HOME/hpeesof/circuit/lib.$SIMARCH
$HPEESOF_DIR/custom/circuit/lib.$SIMARCH
$HPEESOF_DIR/circuit/lib.$SIMARCH
```



Note

For information about \$SIMARCH, see *Determining \$SIMARCH*.

These directories are searched in the order listed. Change the default path by setting the variable EESOF_MODEL_PATH in either of the following files:

```
$HPEESOF_DIR/custom/config/hpeesofsim.cfg
$HOME/hpeesof/config/hpeesofsim.cfg
```

For example (the default setting), see the entry in:

```
$HPEESOF_DIR/config/hpeesofsim.cfg
```

2. Copy the dynamically-loaded device to one of the directories listed in EESOF_MODEL_PATH (see step 1).
3. In the directory where the dynamically-loaded device was copied, the following command must be executed:

```
hpeesofsim -X
```

This will start the simulator, but instead of running a simulation, the current directory will be scanned for dynamically-loaded devices, and an index file (*deviceidx.db*) will be created. Copying a dynamically-loaded device to a directory is not enough. The directory dynamically-loaded device index must also be updated to include the new device. If this is not done, the simulator will be unable to locate the dynamically-loaded device. No simulator licenses of any type are required for this.

**Note**

To run *hpeesofsim*, you must have *\$HPEESOF_DIR/bin* in *\$PATH*, and you must also have set the appropriate environment variables to tell your system about the ADS shared libraries/DLLs and device libraries. For information about setting these environment variables, refer to *ADS Simulator Input Syntax (cktsim)* in *Using Circuit Simulators (cktsim)*.

About User-Compiled Model Code

An unlimited number of user-defined elements in any number of C modules can be written, compiled and linked to your circuit simulator program. Linear elements can have up to 99 external pins, while nonlinear and transient elements can have unlimited number of external pins and internal nodes.

An element without external pins is treated as a Model Form that has no electrical characteristics. Other elements can refer to this Model Form to obtain parameter values.

Element names and parameter keywords are limited to alphanumeric characters and the underscore character. Names cannot begin with a numeric character. In addition, a leading underscore is not recommended as this can interfere with the built-in variables. Any number of parameters of arbitrary type (integer, real, string or Model Form reference) are allowed for each element. A Model Form reference can refer to either a built-in or a user-defined Model Form.

For use in DC and frequency-domain simulations, an element can have either a linear or nonlinear model. Either type of element can have a transient model for use in a Transient simulation.

Linear and noise analysis responses of elements are computed in the frequency domain. The linear response can be computed either in complex scattering matrix or admittance matrix form. The noise response must be computed in complex current correlation matrix form. A user-defined linear element can call most any existing linear element to obtain its response.

Pre- and post-analysis entry points during program execution are provided to enable such calls and to perform special operations, such as data file reading and memory allocation/de-allocation.

Nonlinear element response is computed in the time domain at a sequence of time samples. Time-to-frequency transformations are computed in the circuit simulator engine and are transparent to the user. Element response is characterized by a set of instantaneous (nonlinear) currents out of each pin, nonlinear charges at each pin and their respective derivatives, all determined by applied pin voltages. The user's computation functions cannot call other nonlinear elements for their responses. Element models with time-delay dependencies are supported.

Transient element response is computed in the time domain. Element response is characterized by a set of instantaneous (nonlinear) currents out of each pin, nonlinear charges at each pin and their respective derivatives, all as determined by applied pin voltages. Transient computation functions cannot call other elements (except for ideal resistors, capacitors, inductors and transmission lines) for their responses.

Convolution element response can be computed in two ways. One way is to use a linear model frequency response function so that the circuit simulation engine can compute the time-domain impulse response. Alternatively, specific nonlinear transient element response code can be used.

User-Compiled Model data structures and APIs are explained here. For data structure diagrams, and User-Defined Models API list, refer to *User-Defined Models API Reference* (modbuild).

Macro Definitions

Interfacing to the simulator code requires the use of certain ADS defined public C symbols in user-defined element modules. The remainder of this section describes the supplied `userdefs.h` file that contains these symbols (macros, interface data structure typedefs, and function declarations). Note that the Model Development Kit interface will automatically generate most of these functions and that the header file will automatically be included.

Success or failure of a typical interface function call is determined by its return value, 1 for success and 0 for failure. Therefore, the 'boolean' typedef and these macros are provided. Although this boolean type is integer-valued, only `TRUE` and `FALSE` values should be associated with it.

```
#define FALSE      0
#define false      0
#define TRUE       1
#define true       1
typedef int boolean;
```

Four macros define the Boltzmann constant (Joules/Kelvin), the charge of an electron (Coulombs), the negative of absolute zero temperature (Celsius), and the standard noise reference temperature (Kelvin). The noise-current correlation parameters returned by an element's noise analysis function must be normalized to `FOUR_K_TO` --these parameters have admittance dimensions.

```
/* define some physical constants */
#define BOLTZ      1.380658e-23
#define CHARGE     1.60217733e-19
#define CTOK       273.15
#define NOISE_REF_TEMP 2 90.0 /* standard noise reference temperature, in Kelvin */
#define FOUR_K_TO (4.0*BOLTZ*NOISE_REF_TEMP) /* noise normalization 4kToB, B=1 Hz */
```

This macro obtains the number of items in an array definition at compile time.

```
#define siz(thing) (sizeof(thing)/sizeof(*thing))
```

For clarity, an argument passed by reference can be prefixed by one of these macros in an ANSI function definition and prototype declaration.

```
#define IN /* input argument to function */
#define OUT /* output argument: modified/set by function */
#define INOUT /* argument used and modified by function */
#define UNUSED /* unused argument */
```

The following C macros replace corresponding Series IV functions, which returned scale

factors to convert a parameter value to SI. In ADS, parameter data are always considered to be in SI; hence these macros always return 1.0, and are meant only for Series IV compatibility.

```
#define get_funit(eeElemInst) 1.0 /* freq unit */
#define get_runit(eeElemInst) 1.0 /* resistance unit */
#define get_gunit(eeElemInst) 1.0 /* conductance */
#define get_lunit(eeElemInst) 1.0 /* inductance */
#define get_cunit(eeElemInst) 1.0 /* capacitance */
#define get_lenunit(eeElemInst) 1.0 /* length unit */
#define get_tunit(eeElemInst) 1.0 /* time unit */
#define get_angunit(eeElemInst) 1.0 /* angle unit */
#define get_curunit(eeElemInst) 1.0 /* current unit */
#define get_volunit(eeElemInst) 1.0 /* voltage unit */
#define get_watt(eeElemInst, power) (power) /* power unit */
```

Data Structures

COMPLEX

Linear response modeled in the frequency domain is complex, so the `COMPLEX` type is used for admittance (Y), scattering (S), and noise current-correlation parameters.

```
typedef struct
{
    double real;
    double imag;
}
COMPLEX;
```

DataTypeE

Each element parameter has a specific type.

```
typedef enum
{
    NO_data = -1,      /* unspecified */
    REAL_data = 0,
    INT_data = 1,
    MTRL_data = 2,     /* for parameter referring to an instance */
    STRG_data = 3,
    CMPLX_data = 4,
    INT_VECTOR_data,
    REAL_VECTOR_data,
    CMPLX_VECTOR_data,
    REPEAT_param
}
DataTypeE;
```

UserParamType

Each element parameter definition consists of a keyword string and type.

```
typedef struct {
    char * keyword;
```

```

    DataTypeE  dataType;
}
UserParamType;

```

UserParamData

The parameter values of an item are obtained in an array of the `UserParamData` type. `dataType` is the discriminator tag to determine the actual value of the union. For example, if it is `MTRL_data`, `value.eeElemInst` will refer to a substrate or model form.

```

typedef struct
{
    DataTypeE  dataType;
    union
    {
        double dVal; /* for REAL_data */
        int iVal; /* for INT_data */
        void *eeElemInst; /* for MTRL_data */
        char *strg; /* for STRG_data */
        void *data; /* for vector data or repeated parameter */
    }value;
} UserParamData;

```

NParType

This type can be used specifically for 2-port elements if the conventional 2-port noise parameters are available.

```

typedef struct
{
    double nFmin; /* Noise Figure (dB) */
    double magGamma; /* opt. source Gamma magnitude */
    double angGamma; /* opt. source Gamma phase(radians) */
    double rnEff; /* Effective normalized noise resistance */
    double rNorm; /* Normalizing resistance (ohms) */
} NParType;

```

UserElemDef

`UserElemDef` is the most important data structure which contains the model information and function pointers.

```

typedef struct _UserElemDef UserElemDef;
struct _UserElemDef
{
    char *name; /* Element name. Not to exceed 8 characters */
    int numExtNodes; /* Number of external nodes, max. 20 for linear element */
    int numPars; /* Number of parameters for this element */
    UserParamType *params; /* parameter array */
    /* pre-analysis function: called after element item parsed successfully */
    boolean (*pre_analysis)(INOUT UserInstDef *pInst);

    /* Linear analysis function: called once for each new frequency point.
     * Must return the item's admittance matrix in yPar array.
     * Only used by linear element. For nonlinear element, the pointer should be NULL */
    boolean (*compute_y)(IN UserInstDef *pInst, IN double omega, OUT COMPLEX *yPar);
    /* Linear noise-analysis function: called once for each new frequency point.
     * Must return the item's noise-current correlation admittance, normalized to
     * FOUR_K_TO in nCor array. NULL if noiseless */
    boolean (*compute_n)(IN UserInstDef *pInst, IN double omega, IN COMPLEX *yPar, OUT COMPLEX
nCor);

```

User-Defined Models

```
/* post-analysis: called before the simulation finishes. It is only used by command line
simulations, not ADS simulations */
boolean (*post_analysis)(INOUT UserInstDef *pInst);
UserNonLinDef *devDef; /* User's nonlinear device definition (NULL if linear) */
struct _SeniorType *seniorInfo; /* Senior user defined type and data (arbitrary) */
UserTranDef *tranDef; /* User's transient definition; NULL if none */
/* The following information is new beginning with ADS 2003C */
/* If the following is defined in _UserElemDef, load_elements2() should be called from the
model
booting function boot_senior_MyModel() instead of load_elements() */
int version;
/* The modify_param function is called when any of the device parameter values change */
BOOLEAN (*modify_param)(INOUT UserInstDef *pInst);
};
```

UserNonLinDef

A nonlinear element must contain additional device information in a static area of type `UserNonLinDef` (described later); the pointer `UserElemDef->devDef` must point to it. The `seniorInfo` field is of arbitrary type, and can be used for any extra user-defined data/description that is of no concern to the simulator.

```
typedef struct _UserNonLinDef UserNonLinDef;
struct _UserNonLinDef
{
    int numIntNodes; /* # internal nodes of device */
    /* Evaluate linear part (Y-pars) of device model */
    boolean (*analyze_lin)(IN UserInstDef *pInst, IN double omega)
    /* Evaluate nonlinear part of device model:
    * nonlinear current out of each pin, nonlinear charge at each pin
    * derivative (w.r.t. pin voltage) of each nonlinear pin current, i.e. nonlinear conductance
    g,
    * derivative (w.r.t. pin voltage) of each nonlinear pin charge, i.e. nonlinear capacitance c
    */
    boolean (*analyze_nl)(IN UserInstDef *pInst, double *pinVoltage);
    /* Evaluate small-signal AC model: compute total (linear+linearized) Y-pars of device */
    boolean (*analyze_ac)(IN UserInstDef *pInst, IN double *pinVoltage, IN double omega);
    struct _SeniorModel *modelDef; /* user-defined Senior MODEL (arbitrary) */
    /* Evaluate bias-dependent linear noise model: compute total (linear+linearized) noise-current
    correlation parameters (normalized to FOUR_K_TO, siemens) of device */
    boolean (*analyze_ac_n)(IN UserInstDef *pInst, IN double *pinVoltage, IN double omega);
};
```

UserTranDef

A transient response for an element can be defined in a structure of type `UserTranDef` (described later); the pointer `UserElemDef->tranDef` must point to the structure. A transient response function can be defined for either a linear or nonlinear element.

`numIntNodes` is an arbitrary number of nodes internal to the element. In its model, the element must compute the contributions at all its pins, which are ordered and numbered (starting at zero) with the external pins first, followed by internal pins. If a `UserNonLinDef` type is defined for the element, the `numIntNodes` in that structure must match this definition.

Special routines are available to simplify the use of ideal resistors, capacitors, inductors, and transmission lines within a transient element. For the circuit simulator engine to perform the appropriate allocations, the number of these elements (except resistors) must be predefined using `numCaps`, `numInds`, and `numTlines`.

The `analyze_tr` function must compute and load the instantaneous time domain response of the element, using the element's pin voltages as inputs. The passed array `pinVoltage` contains the instantaneous voltages at both the external and internal pins.

If P is the total number of pin voltages, formulate nonlinear current and charges at each pin n as follows:

$r_n(t) = f(v_0(t), v_1(t), \dots, v_{P-1}(t))$ where r_n is the current out of the pin or charge response. These responses and their voltage derivatives (nonlinear conductances and capacitances) must be computed and loaded using the `add_tr_iq` and `add_tr_gc` functions, respectively. Note that the derivatives are used to help converge to a solution, therefore the simulator may reach a solution even if they are not exact. However, under certain simulation conditions, inexact derivatives may cause convergence problems. Also, for convergence reasons, they should be continuous.

The `fix_tr` function is called just before transient analysis begins. Its only purpose is to set up ideal transmission lines for the user. Using the `add_tr_tline` function, transmission line nodes and physical constants are defined. Once the transmission line is defined here, time-domain analysis of it is performed automatically without any further action by the user in the `analyze_tr` function.

```
typedef struct _UserTranDef UserTranDef;
struct _UserTranDef
{
    int      numIntNodes; /* internal nodes of device */
    int      numCaps;     /* number of explicit capacitors */
    int      numInds;     /* number of explicit inductors */
    int      numTlines;   /* number of explicit transmission lines */
    boolean useConvolution; /* use linear response for convolution */
    /* Evaluate transient model
     * nonlinear currents out of each pin,
     * nonlinear charge at each pin,
     * derivative (w.r.t. pin voltage) of each nonlinear pin current, i.e. nonlinear conductance
g,
     * derivative (w.r.t. pin voltage) of each nonlinear pin charge, i.e. nonlinear capacitance
c
     */
    boolean (*analyze_tr)(IN UserInstDef *pInst, IN double *pinVoltage);
    /* Pre-transient analysis routine used to allocate, compute and connect ideal transmission
    lines */
    boolean (*fix_tr)(IN UserInstDef *pInst);
};
```

UserInstDef

Each user-defined item placed in a design is represented in the ADS Simulator by the item type `UserInstDef`. All the fields, except `seniorData`, in an item are set up by ADS Simulator and must not be changed. `seniorData` can refer to arbitrary data and is meant to be managed by user code exclusively.

```
typedef struct _UserInstDef UserInstDef;
struct _UserInstDef
{
    char *tag; /* item name */
    UserElemDef *userDef; /* access to user-element definition */
    UserParamData *pData; /* item's parameters */
};
```


User-Defined Models

```
void *eeElemInst; /* EEsof's element item */
void *eeDevInst; /* EEsof's nonlinear device item */
void *seniorData; /* data allocated/managed/used only by Senior module (arbitrary) */
};
```

Function Pointers for analysis

pre_analysis

Each user-element definition is of the `UserElemDef` type. The `pre_analysis` function is useful for one-time operations such as parameter type checking, allocating memory, and reading data files. This routine is called for all types of analysis.

```
boolean (*pre_analysis)(INOUT UserInstDef *pInst);
```

post_analysis

Note that a nonlinear or parametric subnetwork instantiation will be flattened (expanded) in the parent network. If there are two or more uses of a given subnetwork, each occurrence will result in the pre-analysis function (and post-analysis function) being called. The function must be written to properly manage such actions as reading data files and allocating memory.

```
boolean (*post_analysis)(INOUT UserInstDef *pInst);
```

compute_y

The `compute_y` function must be defined for linear models. It loads the nodal admittance matrix parameters at frequency ω radians/sec into the passed `yPar` array. When frequency is zero, a value of 1 is passed to ω . This function can call `ee_compute_y` (described later) to use another element's admittance parameters.

```
boolean (*compute_y)(IN UserInstDef *pInst, IN double omega, OUT COMPLEX *yPar);
```

compute_n

The `compute_n` function is used by linear model noise analysis. It must load the normalized nodal noise current correlation parameters (Siemens, normalized to `FOUR_K_TO`) into the passed `nCor` array at frequency ω radians/sec and the element admittance parameters, `yPar`. It can call `ee_compute_n` (described later) to make use of another element's admittance and noise correlation matrices.

```
boolean (*compute_n)(IN UserInstDef *pInst, IN double omega, IN COMPLEX *yPar, OUT COMPLEX *nCor);
```

analyze_lin and analyze_nl

`analyze_lin` must be defined for nonlinear models. It loads only the linear part (complex admittances) of the nonlinear element in the frequency domain. Each admittance must be loaded by calling the primitive `add_lin_y` function. For a branch admittance between nodes (i, j), 4 calls are needed: +Y for (i, i), (j, j) and -Y for (i, j) and (j, i). `analyze_lin` can use `ee_compute_y` to take advantage of pre-existing linear elements. `analyze_nl` must compute and load the nonlinear response, using the element's pin voltages as input. The passed array `pinVoltage` contains instantaneous values; however, delayed voltage differences can be obtained using the `get_delay_v` function.

```
boolean (*analyze_lin)(IN UserInstDef *pInst, IN double omega);
```

If P is the total number of pin voltages, formulate non-zero nonlinear current and charge at each pin n as follows:

$r_n(t) = f(v_0(t), v_1(t), \dots, v_{P-1}(t), v_k(t-*k), v_l(t-*l), \dots)$ where r_n is the pin current or charge response,

*k, *l... are ideal delays, independent of the voltages. These responses and their derivatives with respect to voltage (nonlinear conductances, capacitances) must be computed and loaded using the `add_nl_iq` and `add_nl_gc` functions, respectively. Note that the derivatives help the simulator to converge to a solution, but do not affect the steady-state nonlinear response-therefore they may work even if not exact. However, under certain simulation conditions in-exact derivatives may cause convergence problems. However, for noise analysis they should be accurate, and for convergence they should be continuous.

```
boolean (*analyze_nl)(IN UserInstDef *pInst, double *pinVoltage);
```

analyze_ac

In a linear simulation, a nonlinear element must contribute its small-signal linearized response; this is done through the `analyze_ac` function. The linear part can be loaded by calling the element's `analyze_lin` function. The linearized part is just the nonlinear conductances and capacitances computed above simply converted to admittances at angular frequency `omega` and loaded into the circuit matrix using `add_lin_y`.

```
BOOLEAN (*analyze_ac)(IN UserInstDef *pInst, IN double *pinVoltage, IN double omega);
```

analyze_ac_n

Noise contribution of a nonlinear element in a linear simulation is added through the `analyze_ac_n` function. The linear and linearized noise correlation parameters are loaded

using the primitive `add_lin_n` function. The linearized portion can include shot, flicker, and burst noise contributions.

The `modelDef` field is of arbitrary type and can be used for any extra user-defined nonlinear model data/description that is of no concern to the simulator.

```
BOOLEAN (*analyze_ac_n)(IN UserInstDef *pInst, IN double *pinVoltage, IN double omega);
```

modify_param

The `modify_param` function is called only when a model parameter value changes. The computation of parameter dependent user data can be performed here. This improves the performance of the model parameter sweep or optimization.

If the function exists as a member of the `struct _UserElemDef`, the device loading function `load_elements2()` should be called from `boot_senior_MyModel()` instead of `load_elements()`.

```
BOOLEAN (*modify_param)(INOUT UserInstDef *pInst);
```

Defining Model with Variable Number of External Nodes

When "NumExtNodes" parameter is defined, the user-defined model can accept variable number of nodes.

The function below can be used to get number of external node.

```
int get_ucm_num_external_nodes( const UserInstDef* userInst );
```

Even though there is no need to duplicate the implementation for each node configuration, symbol and parameter AEL definition still needs to be defined for each node configuration individually.

For example, Transmission line `My_TLIN2`(2-terminal) and `My_TLIN4`(4-terminal) share the same model implementation. The model name is "My_TLIN" in the model implementation. `My_TLIN2` and `My_TLIN4` need to have their own symbol definitions and parameter AEL definitions. `My_TLIN2`'s parameter definition is like:

```
create_item( "My_TLIN2",    // component name
            "My ideal 2-Terminal Transmission Line", // description
            "My_TLIN_",    // prefix for the instance name
            0,             // attribute
            -1,            // priority
            "Component Parameters", // component dialog name
            NULL,          // Dialog data
            "ComponentNetlistFmt" // netlist format
            "My_TLIN",     // component netlist name
            "ComponentAnnotFmt", // display format
```

User-Defined Models

```
"" , // Symbol name
0 , // no artwork
NULL , // no artwork data
0 , // no extra attribute
create_parm( "NumExtNodes", "Number of external nodes", PARM_INT|PARM_NOT_EDITED,
             "StdFormSet", -1, prm( "StdForm", "2" ) ),
// ... more create_parm( ) for other parameters
)
```

Note:

- Component netlist name should be "My_TLIN" here, the same as the model name in the implementation.
- "NumExtNodes" need to be a model parameter. It is highly recommended to set the default value to the number of nodes and set PARM_NOT_EDITED attribute to prevent it from being modified.

Referencing Data Items

A user-defined element parameter can be a reference to an ADS or a User-Defined item.

As an example, if you are creating a microstrip element and need an MSUB reference, the third parameter, for example, can be "MSUB" of data type "MTRL_data", the array entry `userInst->pData[2]` will be such that `pData[2].value.eeElemInst` points to the referred MSUB item in the circuit. The MSUB parameters can then be obtained through a `get_params` call:

```
get_params(userInst->pData[2].value.eeElemInst, mData)
```

This will copy the *MSUB parameters* (`ccdlist`) into `mData`, which is an array of [UserParamData](#) elements:

User-Defined Models

Index	Name	Description	Units
0	Er	Relative dielectric constant	
1	Mur	Relative permeability.	
2	H	Substrate thickness.	mil
3	Hu	Cover height.	mil
4	T	Conductor thickness.	mil
5	Cond	Conductor conductivity.	Siemens/meter
6	TanD	Dielectric Loss Tangent.	
7	Rough	Conductor surface roughness.	
8	RoughnessModel	Conductor surface roughness model. 1: Hammerstad 2: Multi-level hemisphere (default).	
9	Bbase	Conductor surface roughness: tooth base width.	mil
10	Dpeaks	Conductor surface roughness: distances between tooth peaks.	mil
11	L2Rough	Conductor surface roughness: height protrusions of level 2.	mil
12	L2Bbase	Conductor surface roughness: tooth base width of level 2.	mil
13	L2Dpeaks	Conductor surface roughness: distances between tooth peaks of level 2.	mil
14	L3Rough	Conductor surface roughness: height protrusions of level 3.	mil
15	L3Bbase	Conductor surface roughness: tooth base width of level 3.	mil
16	L3Dpeaks	Conductor surface roughness: distances between tooth peaks of level 3.	mil
17	<i>reserved</i>		
18	FreqForEpsrTanD	Frequency at which Er and TanD are measured.	Hertz
19	DielectricLossModel	Dielectric loss model 0: Frequency independent 1: Svensson/Djordjevic (default).	
20	HighFreqForTanD	High end frequency in the Svensson/Djordjevic model.	Hertz
21	LowFreqForTanD	Low end frequency in the Svensson/Djordjevic model.	Hertz

into `mData[0...21]` locations. The `mData` array must be dimensioned large enough to hold all the referenced item's parameters (22, in the case of MSUB parameters). If a parameter value is not set or available, the ``dataTypeE'` enum value will be `NO_data`.



Important Note

Substrates, such as MSUB, can contain a differing number of parameters mentioned above, especially between different versions of ADS. Because of this, you must allocate the `mData` array to 22 elements or larger. However, as ADS does not currently contain a method of determining the minimum required size for `mData`, it is recommended that you allocate an `mData` array with at least 100 elements (ADS 2011 requires a minimum of 22 elements, but ADS 2009UR1 required only 13 elements). This should be large enough to hold all substrate parameters, now and in the future. Note that, if you do not allocate a sufficiently large `mData` array, your code may cause the simulator to abort (possibly with a "Segmentation violation" error).

**Note**

The above MSUB array definition applies to ADS 2011.01 and later. For ADS 2009UR1 and earlier, only the first eight items are guaranteed to exist, as some of the above parameters exist only in ADS 2011.01 and later. Attempting to access elements 8 and up in ADS 2009UR1 and older is not guaranteed to work. However, in ADS 2009UR1, you must still provide an `mData` array of size 13 or larger when calling `get_params` to obtain MSUB parameters.

If the referenced item which is an instance of a user-defined model, then you can obtain a pointer to the user item as follows:

```
refInst = get_user_inst( modelInstName )
```

The function `get_user_inst` will return `NULL` if the passed argument is not a user-defined item. An example can be found in "Res" model defined in `examples/Tutorial/UserCompileModel_wrk`.

Querying User-Defined Element Parameters

For user-defined element parameters, supported data types are integer, real, complex, string, integer array, real array and complex array.

Parameter can have more than one entry. For example, `Freq[1]` can be used for fundamental frequency, and `Freq[2]` can be used for frequency at 2nd harmonic. This kind of parameter is called repeated parameter. The attribute for repeat parameter is defined in the model parameter definition `itemdef.ael` file by `create_parm()` function. The `PARM_REPEATED` bit has been set in the third argument of the `create_parm()` function. For example, "Freq" parameter:

```
create_parm("Freq","Frequency",PARM_REPEATED|PARM_REAL,"StdFormSet",-1,prm(""),
prm("StdForm","1e9"))))
```

Parameter value can be an array. "list" function is used to define array value. For example, `Coefficient=list(0.1, 0.2, 0.3)`.

Parameter values are stored in `userInst->pData[paramIndex].value` which is a union. Parameter data type is needed to access union data properly. The following interfaces are provided for easy parameter value access. Please note in some old UserCompiled model implementations, macros are defined to access parameter values. This approach doesn't work for repeated parameter and parameter with array value. Using parameter query interfaces listed below is highly recommended. Examples are provided in `example/Tutorial/UserCompiledModel_wrk`.

The function below returns number of parameters.

```
int get_ucm_num_of_params( const UserInstDef *userInst );
```

The function below returns the pointer to the parameter by parameter index, and repeat index if the parameter is repeated parameter. Note that both parameter index and repeat index start from zero. For example, `Freq` is the first parameter, then for `Freq[1]`, parameter index is 0, and repeat index is 0. For non-repeated parameter, please set

repeat index to -1.

```
const SENIOR_USER_DATA* get_ucm_param_ptr( const UserInstDef* userInst,
                                           int paramIndex, char* errorMsg, int repeatIndex );
```

The function below returns parameter name. Note that parameter index starts from 0.

```
const char* get_ucm_param_name( const UserInstDef* userInst, int paramIndex );
```

The function below returns true if the parameter is repeated parameter, otherwise it returns false.

```
BOOLEAN is_ucm_repeat_param( const UserInstDef* userInst, int paramIndex );
```

The function below returns the number of entries of the repeated parameter. For example we have Freq[1], and Freq[2], then the number of repeats is 2.

```
int get_ucm_param_num_repeats( const UserInstDef* userInst, int paramIndex);
```

The function below returns parameter data type.

```
DataTypeE get_ucm_param_data_type( const SENIOR_USER_DATA *param );
```

Functions below can be used to query parameter with single value. Using corresponding function based on parameter data type. "param", the pointer to the parameter can be obtained from get_ucm_param_ptr(). Before using parameter value, please check "status" in case the query fails for a reason. If the query fails, the error message is returned in errorMsg.

```
int get_ucm_param_int_value( const SENIOR_USER_DATA *param, BOOLEAN* status, char* errorMsg );
double get_ucm_param_real_value( const SENIOR_USER_DATA *param, BOOLEAN* status, char* errorMsg );
Complex get_ucm_param_complex_value( const SENIOR_USER_DATA *param, BOOLEAN* status, char*
errorMsg );
const char* get_ucm_param_string_value( const SENIOR_USER_DATA *param, BOOLEAN* status, char*
errorMsg );
```

The function below returns parameter value array size. If the parameter value is not a vector, zero is returned. "param", the pointer to the parameter can be obtained from get_ucm_param_ptr().

```
int get_ucm_param_vector_size( const SENIOR_USER_DATA *param );
```

Functions below can be used to query parameter with vector value. Using corresponding function based on parameter data type. "param", the pointer to the parameter can be obtained from get_ucm_param_ptr(). "index" is the index in the vector and it starts from 0. Before using parameter value, please check "status" in case the query fails for a reason. If the query fails, the error message is returned in errorMsg.

```
int get_ucm_param_vector_int_value( const SENIOR_USER_DATA *param, int index, BOOLEAN* status,
char* errorMsg );
int get_ucm_param_vector_real_value( const SENIOR_USER_DATA *param, int index, BOOLEAN* status,
char* errorMsg );
int get_ucm_param_vector_complex_value( const SENIOR_USER_DATA *param, int index, BOOLEAN*
status, char* errorMsg );
```

The function outputs each parameter value to *fp. "param", the pointer to the parameter can be obtained from get_ucm_param_ptr(). The function is provided for debug purpose.

```
void print_ucm_param_value( FILE *fp, const SENIOR_USER_DATA *param );
```

The get_params function, below, loads parameter values of the item whose name is defined by eeElemInst into pData, which must be big enough to store all parameters. It is used to obtain other referenced item (such as model substrate) parameters. It returns TRUE if successful, FALSE otherwise. It doesn't work for repeated parameter or parameter with vector value.

```
BOOLEAN get_params (IN void*eeElemInst, OUT UserParamData *pData);
```

The dump_params function, below, obsolete, prints out the instance eeElemInst's parameter names and values to stderr. This function should only be used for debugging purposes. The function returns TRUE if successful, FALSE otherwise. It doesn't work for repeated parameter or parameter with vector value.

```
BOOLEAN dump_params(IN void *eeElemInst);
```

Displaying Error/Warning Messages

You can flag errors within a function in a user-defined element module and send messages to the Simulation/Synthesis panel. You can also write helpful status and debug messages to the Status/Summary panel. The following functions can be used for sending the message to respective locations:

```
extern void send_error_to_scn (char ) / writes message to Errors/Warnings panel*/
```

```
extern void send_info_to_scn (char *) /*writes message to Status/Progress panel */
```

The argument of these functions is a character pointer that is the error message string.

Examples:

```
send_error_to_scn("divide-by-zero condition detected");
send_info_to_scn("value of X falls outside its valid range");
```


Booting All Elements in a User-Defined Element File

In order to keep the code modular, each user-defined element file can contain at most a single external/public symbol definition; this is the booting function, usually named `boot_abc` for a module named `abc.c`:

```
boolean boot_abc(void)
```

This function is called once per module-at program bootup. If the ModelBuilder interface is used, only one model per file is allowed. However, multiple files can be combined into one larger module by the user. The call to boot the module must be included in the self-documented `userindx.c` file at the appropriate location. The module's file name must be added to the `user.mak USER_C_SRCS` definition.

All user-defined elements in the module can be defined in a static (with module-scope) `UserElemDef` array and booted by calling the provided function `load_elements()` or `load_elements2()` from `boot_abc`.

The `load_elements()` function should be used when the version and (`*modify_param`) are not defined as members of the `struct_UserElemDef`.

```
extern boolean load_elements (UserElemDef *userElem, int numElem);
```

The `load_elements2()` function was introduced after ADS 2003C. This function should be used when the version and (`*modify_param`) are defined as members of the `struct_UserElemDef`.

```
extern boolean load_elements2 (UserElemDef *userElem, int numElem);
```

If necessary, you can include code for technology/data file reading, as well as for automatic AEL generation, in the boot function.

Using Built-In ADS Linear Elements in User-Defined Elements

A user-defined element can call an ADS linear element to obtain the latter's COMPLEX Y and noise-correlation parameters. However, nonlinear devices, model items, and independent sources cannot be called in a user-defined element module. The relevant functions in the interface to support this feature are described below:

ee_pre_analysis

This function is usually called from the user-defined element's `pre_analysis` function. It returns a pointer to an allocated ADS item if successful, `NULL` otherwise. This pointer must be saved (possibly with the user-defined element item, in its `seniorData` field) and passed to `ee_compute_y` or `ee_compute_n`.

```
extern void *ee_pre_analysis (char *elName, UserParamData *pData);
```

ee_compute_y

This function allows access to Advanced Design System elements for linear analysis. Note that parameter data `pData` must be supplied in SI units, where applicable. They return `TRUE` if successful, `FALSE` otherwise. To determine parameter order, execute the simulator binary (`hpeesofsim`) using the **-h** flag and the name of the parameter (e.g., `$HPEESOF_DIR/bin/ hpeesofsim -h MLIN`).

The function below obtains $N \times N$ COMPLEX Y-parameters of the N-node (excluding ground) ADS element item in the user-supplied `yPar` array at frequency `omega` radians/sec. It returns `TRUE` if successful, `FALSE` otherwise.

```
extern boolean ee_compute_y (void *eeElemInst, UserParamData *pData, double omega, COMPLEX *yPar);
```

ee_compute_n

This function allows access to Advanced Design System elements for noise analysis. Note that parameter data `pData` must be supplied in SI units, where applicable. They return `TRUE` if successful, `FALSE` otherwise.

The function obtains the $N \times N$ COMPLEX Noise correlation matrix parameters, given `omega` and the $N \times N$ COMPLEX Y-pars. It returns `TRUE` if successful, `FALSE` otherwise.

```
extern boolean ee_compute_n (void *eeElemInst, UserParamData *pData, double omega, COMPLEX *yPar, COMPLEX *nCor);
```

ee_post_analysis

```
extern boolean ee_post_analysis (void *eeElemInst);
```

Other Utility APIs

The `get_temperature` function, below, returns the value of the ADS global variable *temp* in kelvin.

```
extern double get_temperature(void);
```

These functions are useful to indicate program status in various stages of execution, such as during module boot-up, element analyses, and pre- or post-analysis.

```
extern void send_info_to_scn (IN char *msg); /* write msg to Status/Progress window */
extern void send_error_to_scn (IN char *msg); /* write msg to Errors/Warnings window */
```

In nonlinear analyses, for each set of independent input values (bias, frequency, power, or swept variable), ADS simulator attempts to find the steady state solution iteratively. In each iteration, nonlinear parts of all element items, including user-defined items, are evaluated. This function returns `TRUE` whenever the first iteration is in progress. It is most

useful for parameter range checking, which is sufficient to do at the first iteration.

```
extern boolean first_iteration (void);
```

This function returns `TRUE` whenever the circuit is being analyzed at the first point in a frequency plan. Note that this can happen many times in one simulation command for example, if there is another swept variable, or if an optimization/yield analysis is requested.

If a one-time-only operation is to be performed per circuit, the `pre_analysis` function is recommended instead of this function.

```
extern boolean first_frequency (void);
```

The function below can be used to find the location of a data file based on the ADS variable `SIM_FILE_PATH` which is defined in the `hpeesofsim.cfg` configuration file. The function returns the file name together with its absolute path if the file is found. Otherwise the function returns `NULL`.

```
extern const char* locate_data_file( const char* fileName );
```

The function below computes the normalized complex noise correlation matrix for a passive element, given its Y-pars, operating temperature and number of pins.

```
extern boolean passive_noise (IN COMPLEX *yPar, IN double tempC,  
                             IN int numNodes, OUT COMPLEX *nCor);
```

The function below computes the normalized complex noise correlation 2*2 matrix for an active 3-terminal, 2-port element/network, given its Y-pars and measured noise parameters. Note that if `numFloatPins` is 2, the common (reference) third terminal is ground.

```
extern boolean active_noise (IN COMPLEX *yPar, IN NParType *nPar,  
                             int numFloatPins, OUT COMPLEX *nCor);
```

The function below must be called (usually from nonlinear model's `analyze_lin` and `analyze_ac` procedure) to add the linear complex Y-parameter (`iPin`, `jPin`) branch contribution. This call must be done even for linear capacitive branches at DC ($\omega = 0$), this will establish the Jacobian matrix entry location for subsequent non-zero harmonic ω .

```
extern boolean s_y_convert (IN COMPLEX *inPar, OUT COMPLEX *outPar, IN int direction,  
                             IN double rNorm, IN int size);
```

```
extern boolean add_lin_y (INOUT UserInstDef *userInst, IN int iPin,  
                          IN int jPin, IN COMPLEX y);
```

The function below must be called (from nonlinear model's `analyze_ac_n` function) to add

the complex noise-current correlation term `iNcorr` (Siemens, normalized to `FOUR_K_TO`) from the (`iPin`, `jPin`) branch.

```
extern boolean add_lin_n (INOUT UserInstDef *userInst, IN int iPin,
                        IN int jPin, IN COMPLEX iNcorr);
```

The function below must be called (from nonlinear model's `analyze_nl` function) to add the nonlinear conductance and capacitance contribution for the (`iPin`, `jPin`) branch.

```
extern boolean add_nl_gc (INOUT UserInstDef *userInst, IN int iPin,
                        IN int jPin, IN double g, IN double c);
```

The function below must be called (from nonlinear model's `analyze_nl` function) to add the nonlinear current and charge contribution at the device pin `iPin`.

```
extern boolean add_nl_iq (INOUT UserInstDef *userInst, IN int iPin,
                        IN double current, IN double charge);
```

The function below can be called (from nonlinear model's `analyze_nl` function) to get tau seconds delayed (`iPin`, `jPin`) voltage difference. Note that tau must not be dependent on device pin voltages--it is an ideal delay.

```
extern boolean get_delay_v (INOUT UserInstDef *userInst, IN int iPin, IN int jPin,
                        IN double tau, OUT double *vDelay);
```

Any transient support function that follows can use ground as a pin by using this special macro:

```
\#define GND \-1
```

The function below can be called (from the transient model's `analyze_tr` function) to obtain the current time value, in seconds, of the transient analysis.

```
extern double get_tr_time (void);
```

The function below must be called (from the transient model's `analyze_tr` function) to add the nonlinear conductance and capacitance contribution for the (`iPin`, `jPin`) branch.

```
extern boolean add_tr_gc (INOUT UserInstDef *userInst, IN int iPin, IN int jPin, IN double g,
                        IN double c);
```

The function below must be called (from the transient model's `analyze_tr` function) to add the nonlinear current and charge contribution at the device pin `iPin`.

```
extern boolean add_tr_iq (INOUT UserInstDef *userInst, IN int iPin,
                        IN double current, IN double charge);
```

The function below can be called (from the transient model's `analyze_tr` function) to add a resistor of `rval` Ohms between `pin1` and `pin2`. The contribution of this resistor need not be included in the other calculated currents, charges and derivatives. If `rval` is less than 10^{-6} , `rval` is set equal to 10^{-6} .

```
extern boolean add_tr_resistor (INOUT UserInstDef *userInst, IN int pin1,
                              IN int pin2, IN double rval);
```

The function below can be called (from the transient model's `analyze_tr` function) to add a capacitor of `cval` Farads between `pin1` and `pin2`. The contribution of this capacitor need not be included in the other calculated currents, charges and derivatives. If `cval` is zero, an open circuit will exist between `pin1` and `pin2`.

```
extern boolean add_tr_capacitor (INOUT UserInstDef *userInst, IN int pin1, IN int pin2,
                                IN double cval);
```

The function below can be called (from the transient model's `analyze_tr` function) to add an inductor of `lval` Henries between `pin1` and `pin2`. The contribution of this inductor need not be included in the other calculated currents, charges and derivatives. If `lval` is zero, a short circuit will exist between `pin1` and `pin2`.

```
extern boolean add_tr_inductor (INOUT UserInstDef *userInst, IN int pin1, IN int pin2,
                               IN double lval);
```

The function below can be called to simplify the work for adding a lossy inductor. One more argument is passed to the function which is the resistance `rval` Ohms of the inductor.

```
extern boolean add_tr_lossy_inductor (INOUT UserInstDef *userInst, IN int pin1, IN int pin2,
                                     IN double rval, IN double lval);
```

The function below can be called to add mutual inductance with coupling coefficient of `kval` between the inductor `ind1` and the inductor `ind2`.

```
extern boolean add_tr_mutual_inductor (INOUT UserInstDef *userInst, IN int ind1, IN int ind2,
                                       IN double kval);
```

`add_tr_inductor ()` or `add_tr_lossy_inductor ()` must be added before the mutual inductance is added. `ind1` and `ind2` are the values returned from `add_tr_inductor ()` or `add_tr_lossy_inductor ()`. `add_tr_inductor ()` and `add_tr_lossy_inductor ()` return a positive integer upon the successful completion. The integer is the inductor index which can be passed to `add_tr_mutual_inductor ()` as an inductor ID. 0 is returned when the function call fails. Here is an example:

User-Defined Models

```
int ind1, ind2;
boolean status = TRUE;
ind1 = add_tr_lossy_inductor(userInst, 0, 2, R1, L1);
ind2 = add_tr_lossy_inductor(userInst, 1, 3, R2, L2);
if( ind1 && ind2 )
status = add_tr_mutual_inductor(userInst, ind1, ind2, K12);
else
status = FALSE;
```

The function below can be called (from the transient model's `fix_tr` function) to add an ideal transmission line. The impedance of the line is `z0` Ohms and the propagation delay time of the line is `td` seconds. The loss parameter is used to describe the voltage attenuation on the line; a loss of 1.0 specifies a lossless line; a loss of 0.5 specifies an attenuation of 6 dB. The time domain simulation of this transmission line will be computed automatically with no further action by the user in the `analyze_tr` function.

```
extern boolean add_tr_tline (INOUT UserInstDef *userInst, IN int pin1,
IN int pin2, IN int pin3, IN int pin4, IN double z0, IN double td,
IN double loss);
```

Creating Linear Circuit Elements

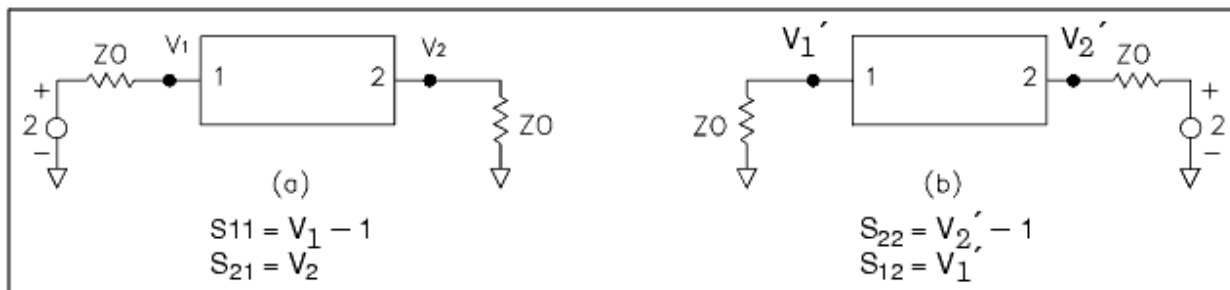
This section describes creating linear elements through the use of examples. A linear element differs from a nonlinear element in that a linear element contains only linear elements while a nonlinear element can contain both linear and nonlinear elements.

Deriving S-Parameter Equations

One way to characterize a circuit element is by its S-parameters. To help you derive the S-parameters, refer to the following book: *Microwave Transistor Amplifiers* by Guillermo Gonzalez (Englewood Cliffs: Prentice-Hall, Inc., 1984).

Begin the process of deriving the S-parameters by examining the circuit configurations shown in the following schematic. Although this example shows 2-port S-parameters, the technique is the same for elements with a greater number of ports.

Figure1: 2-port network



Alternative, but equivalent, expressions for S_{11} and S_{22} are

$$S_{11} = (Z_1 - Z_0)/(Z_1 + Z_0)$$

$$S_{22} = (Z_2 - Z_0)/(Z_2 + Z_0)$$

where

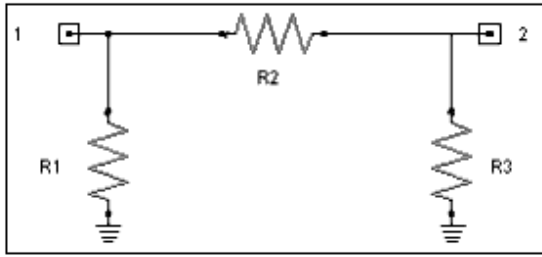
Z_0 is the normalizing impedance for the circuit (usually 50 ohms)

Z_1 is the impedance looking into port 1 when port 2 is terminated with Z_0

Z_2 is the impedance looking into port 2 when port 1 is terminated with Z_0

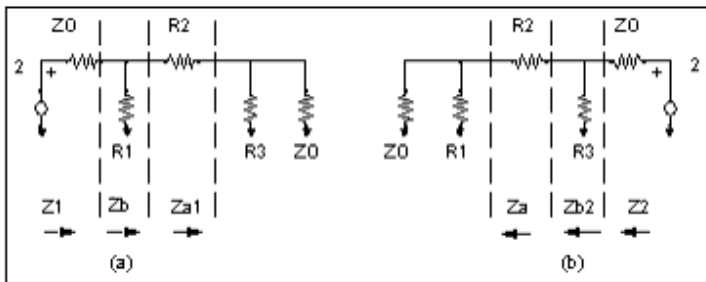
For example, consider a grounded pi-section resistive attenuator as shown in the following schematic.

Figure2: Schematic for pi-section resistive attenuator



Inserting the schematic shown above into the 2-port network results in the following schematic.

Figure3: Resulting circuit schematic



Using the schematic above, the following relations are defined:

$$\begin{aligned} Y_{A1} &= 1.0/R3 + 1.0/Z0 \\ Z_{A1} &= 1.0/Y_{A1} \\ Z_{B1} &= R2 + Z_{A1} \end{aligned}$$

Because the network is symmetrical, the following relations also hold:

$$\begin{aligned} Y_{A2} &= 1.0/R1 + 1.0/Z0 \\ Z_{A2} &= 1.0/Y_{A2} \\ Z_{B2} &= R2 + Z_{A2} \end{aligned}$$

From the definition of Z1 and Z2:

$$\begin{aligned} Z1 &= (R1 \cdot Z_{B1}) / (R1 + Z_{B1}) \\ Z2 &= (R3 \cdot Z_{B2}) / (R3 + Z_{B2}) \end{aligned}$$

The S-parameters are obtained from the following equations:

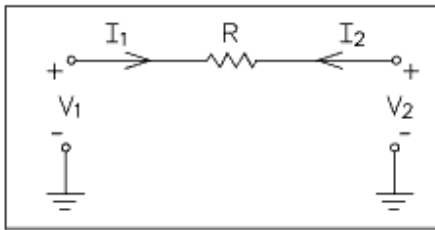
$$\begin{aligned} S_{11} &= (Z1 - Z0) / (Z1 + Z0) \\ S_{22} &= (Z2 - Z0) / (Z2 + Z0) \\ S_{12} &= S_{21} = (2.0 / Z0) / (1.0 / Z_{A1} + 1.0 / Z_{A2} + R2 / (Z_{A1} \cdot Z_{A2})) \end{aligned}$$

These basic equations are sufficient to write the C function for the element.

Deriving Y-Parameter Equations

Y-parameters equations can be used to describe a user-defined element as an alternative to S-parameter equations. The following schematic shows Y-parameters for a resistor connected between two ports; Y-parameter definitions follow the figure.

Figure4: Y-parameters for a 2-port resistor connection



In general,

$$Y_{11} = \left. \frac{I_1}{V_1} \right|_{V_2=0} \quad Y_{21} = \left. \frac{I_2}{V_1} \right|_{V_2=0}$$

$$Y_{12} = \left. \frac{I_1}{V_2} \right|_{V_1=0} \quad Y_{22} = \left. \frac{I_2}{V_2} \right|_{V_1=0}$$

With V_2 equal to zero, $V_1 = I_1 R$, which is also equal to $-I_2 R$. Y_{11} reduces to $1/R$ and Y_{21} to $-1/R$. Setting V_1 to zero, $V_2 = I_2 R = -I_1 R$. The expressions for Y_{22} and Y_{12} are $1/R$ and $-1/R$, respectively. The resultant Y-parameter matrix is:

$$[Y] = \begin{bmatrix} 1/R & -1/R \\ -1/R & 1/R \end{bmatrix}$$

The following code is a portion of the example file:

```

/*****
/
# define EPS 1.0e-8
/*
* This example shows direct Y-parameter loading, instead of S-parameters.
* For some elements, admittance parameters are easier to derive than
* scattering parameters. For a series resistor, the admittance matrix is
* as follows:
*
*      | g  -g |
*      | -g  g |   where g = 1 / R

```

```

*
* ELEMENT U2PD Id n1 n2 R=#
*/
static boolean u2pd_y(
UserInstDef *userInst,
double omega,
COMPLEX *yPar)
{
    double res, cond;
    res = userInst->pData->value.dVal * get_runit(userInst->eeElemInst);
    if (res < EPS)
        res = EPS;
    cond = 1.0 / res;
    yPar[0].real = yPar[3].real = cond;
    yPar[1].real = yPar[2].real = -cond;
    yPar[0].imag = yPar[1].imag = yPar[2].imag = yPar[3].imag = 0.0;
    return TRUE;
}
#undef EPS
/*****
/

```

Coding a Linear Element

Your circuit simulator includes examples of linear user-compiled models. You can follow the same style in your modules. You can define only one model per module. Every model includes a `*_h.c` file, which contains macros, type definitions, and interface function declarations. If you are interested you can study this file to learn how dialog box settings map to the c-code. Note that the file is automatically generated so any changes made directly to the file will be lost.

To create a linear element, perform the following steps:

1. Define the element from the parameters page and define the number of external pins from the Symbol View (accessed from the Model Code tab).
2. Write the function to return the linear response. The linear behavior is characterized by a linear analysis function that you will write; this corresponds to the `compute_y` function pointer in the `UserElemDef` structure (already defined in the template code file):

```

boolean (*compute_y)(UserInstDef *pInst, double omega, COMPLEX
*yPar)

```

This function must return `TRUE` if successful, `FALSE` otherwise. This function should be capable of working at $\omega = 0$, especially if it is used for convolution. You can use Y-parameters directly, or compute S-parameters and call the supplied `s_y_convert` function to obtain Y-parameters:

```

extern boolean s_y_convert(COMPLEX *inPar, COMPLEX *outPar, int
direction, double rNorm, int size)

```

3. Write the function to return the linear noise response. The linear noise behavior is characterized by a noise analysis function; this corresponds to the `compute_n` function pointer (already defined in the template code file):

```
boolean (*compute_n)(UserInstDef *pInst, double omega, COMPLEX
*yPar, COMPLEX *nCor);
```

It must compute the $N \times N$ COMPLEX noise correlation matrix using the passed arguments `omega` and `yPar` array. The Code Options dialog box setting will set this to NULL if the element is noiseless. The function must return TRUE if successful, FALSE otherwise.

Thermal noise generated by a user-defined passive n -port element (where n is between 1 and 20) at some element temperature `tempC` deg. Celsius can be included in the nodal noise analysis of the parent network by calling the provided function:

```
boolean passive_noise(COMPLEX *yPar, double tempC, int numNodes,
COMPLEX *nCor)
```

from the element's `compute_n` function.

For an active 3-terminal 2-port element, if the conventional 2-port noise parameters (minimum noise figure, optimum source reflection coefficient, effective noise resistance) are available through a measured data file, the 2×2 COMPLEX noise correlation matrix required by `compute_n` can be obtained using the provided function:

```
boolean active_noise (COMPLEX *yPar, NParType *nPar, int
numFloatPins, COMPLEX *nCor)
```

`numFloatPins` is either 3 for floating reference pin, or 2 for grounded reference pin. You must fill the noise parameters into the `nPar` structure.

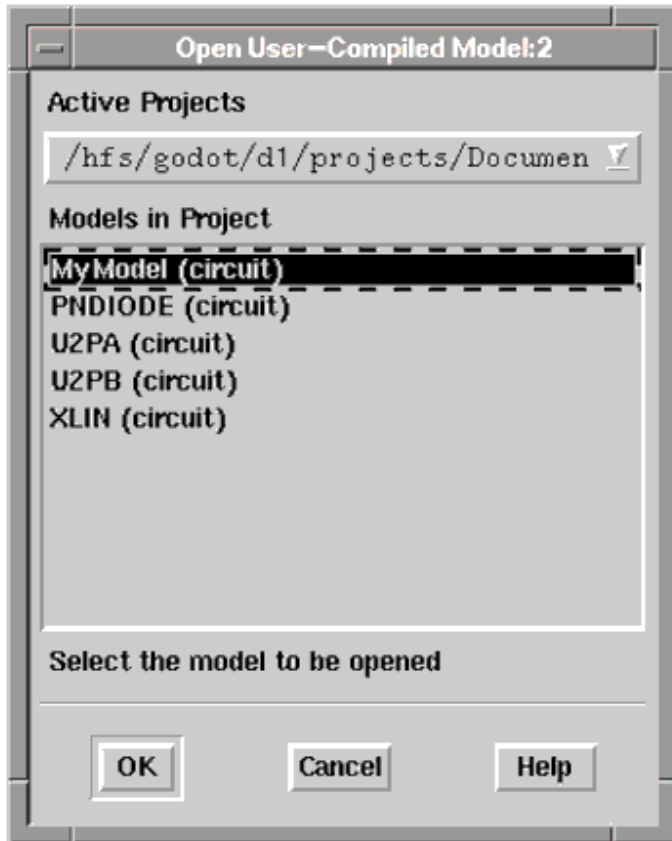
4. If the element needs special pre-analysis processing, such as reading data/technology files, the `pre_analysis` pointer must be set to an appropriate processing function. The Code Options dialog box value will determine whether this pointer is set to NULL or to the `pre_analysis` function. The function must return TRUE if successful, FALSE otherwise.
5. Before the beginning of a new circuit analysis, you must write the function for any cleanup or post-processing required by the element (such as freeing memory or writing an output file) and set the Post-Analysis Function check box in the Code Options dialog. The function must return TRUE if successful, FALSE otherwise.
6. To allow detailed or extra information in your user-defined element definition, the pointer field `seniorData` can be used to point to an arbitrary structure.

Pi-Section Resistive Attenuator

The steps in the preceding section [Coding a Linear Element](#) are described for the grounded pi-section resistive attenuator example U2PA in the following sections.

Element Definition

The array *U2PA* (with static or module scope) defines the parameters of the U2PA element. The c code is automatically generated by the information in the dialog box.



The corresponding c-headers are automatically generated:

```
#define R1_P  userInst->pData[0].value.dVal
#define R2_P  userInst->pData[1].value.dVal
#define R3_P  userInst->pData[2].value.dVal
static UserParamType
U2PA_parms[] =
{
{"R1", REAL_data}, {"R2", REAL_data}, {"R3", REAL_data}
};
static UserElemDef U2PA_ELEMENTS[] =
{
"U2PA", /* modelName */
NUM_EXT_NODES, /* # of external nodes */
siz(U2PA_PARMS), /* # of parameters */
U2PA_PARMS, /* # of parameter structure */
PRE_ANALYSIS_FCN_PTR, /* pre-analysis fcn ptr */
COMPUTE_Y_FCN_PTR, /* Linear model fcn ptr */
COMPUTE_N_FCN_PTR, /* Linear noise model fcn ptr */
POST_ANALYSIS_FCN_PTR, /* post-analysis fcn ptr */
NULL, /* nonlinear structure ptr */
NULL, /* User-defined arb. data structure */
};
```

It is up to the user to write the appropriate code for the `compute_y` and `compute_n` functions.

Defining Variables

Begin by defining the variables and their data types. The S-parameter equations derived in [Resulting circuit schematic](#) provide the basis for the needed variables. The equations are repeated here:

$$\begin{aligned}
 YA1 &= 1.0/R3 + 1.0/ZO \\
 ZA1 &= 1.0/YA1 \\
 ZB1 &= R2 + ZA1 \\
 YA2 &= 1.0/R1 + 1.0/ZO \\
 ZA2 &= 1.0/YA2 \\
 ZB2 &= R2 + ZA2 \\
 Z1 &= (R1 \times ZB1)/(R1 + ZB1) \\
 Z2 &= (R3 \times ZB2)/(R3 + ZB2) \\
 S_{11} &= (Z1 - ZO)/(Z1 + ZO) \\
 S_{22} &= (Z2 - ZO)/(Z2 + ZO) \\
 S_{12} = S_{21} &= (2.0 / ZO)/(1.0 / ZA1 + 1.0 / ZA2 + R2/(ZA1 \bullet ZA2))
 \end{aligned}$$

The resulting declarations are:

```
double YA1, YA2;
double ZA1, ZA2, ZB1, ZB2;
double Z1, Z2;
COMPLEX S[4];
```

Implementing S-Parameter Equations

Implement the equations by performing the following steps:

1. The parameters are available via macro definitions as the parameter name, with an appended `_P`:
`R1 = R1_P;`
`R2 = R2_P;`
`R3 = R3_P;`
2. Include code to check the resistance values and protect against division by zero. In this example, the expressions `YA1` and `YA2` demonstrate the need to check data values. If `R1` or `R3` has a value of zero, a fatal division by zero error condition will result.
 To protect against division by zero, limit the lower value of all input parameters to an arbitrarily low value. For easy use, assign the value to a C macro, for example, `EPS` to mean 10^{-8} .

```
#define EPS 1.0E-8
if (R1 < EPS)
    R1 = EPS;
if (R2 < EPS)
    R2 = EPS;
if (R3 < EPS)
    R3 = EPS;
```

3. Insert code to define the equations. Note that the 2×2 Y-parameters to be returned in the `yPar[0..3]` locations must be in row order, for example, Y_{11} , Y_{12} , Y_{21} and Y_{22} .

```
S[3].imag = S[2].imag = S[1].imag = S[0].imag = 0.0; /* imag part */
YA1 = 1.0 / R3 + 1.0 / ZO;
ZA1 = 1.0 / YA1;
ZB1 = R2 + ZA1;
Z1 = (R1 * ZB1) / (R1 + ZB1);
S[0].real = (Z1 - ZO) / (Z1 + ZO); /* S11 real */
YA2 = 1.0 / R1 + 1.0 / ZO;
ZA2 = 1.0 / YA2;
ZB2 = R2 + ZA2;
Z2 = (R3 * ZB2) / (R3 + ZB2);
S[3].real = (Z2 - ZO) / (Z2 + ZO); /* S22 */
S[2].real = S[1].real = (2.0/ZO) /
(1.0/ZA1 + 1.0/ZA2 + R2/(ZA1 * ZA2));
/* convert S[2x2] -> yPar[2x2] */
return s_y_convert(S, yPar, 1, ZO, 2);
return status;
```

Adding Noise Characteristics

The noise analysis function pointer `compute_n` for passive elements in this example is set to `thermal_n`, which computes thermal noise of the element item at the simulator default temperature of 27.0°C, as shown below.

```
#define STDTEMP 27.0
/*
 * Thermal noise model at default temperature (27.0 deg.C) for any
 * n-terminal linear element
 */
static boolean thermal_n(
    UserInstDef *userInst,
    double omega,
    COMPLEX *yPar,
    COMPLEX *nCorr)
{
    UserElemDef *userDef = userInst->userDef;
    return passive_noise(yPar, STDTEMP, userDef->numExtNodes, nCorr);
}
```

The `passive_noise` function uses the supplied $N \times N$ Y-parameters of an N-terminal element and temperature to compute the $N \times N$ complex noise correlation matrix.

It is possible to compute thermal noise at variable temperatures by adding a temperature parameter, which could be either `REAL_data` or a `MTRL_data` reference, to the element definition.

The next step is compiling and linking the C code. Refer to *Building User-Compiled Analog Models* (modbuild).

Transmission Line Section

This example will show how to derive an S-parameter matrix for a general transmission

line section, then show how to apply this to the case of a coaxial cable. The end result will be an element that can produce an S-parameter matrix given physical dimensions.

Deriving an S-Parameter

The ABCD matrix for a general section of lossless transmission line is:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} \cos\left(\frac{2\pi L}{\lambda}\right) & jZ \sin\left(\frac{2\pi L}{\lambda}\right) \\ j\frac{1}{Z} \sin\left(\frac{2\pi L}{\lambda}\right) & \cos\left(\frac{2\pi L}{\lambda}\right) \end{bmatrix}$$

From *Microwave Transistor Amplifiers* by G. Gonzalez, the conversion from an ABCD to an S-matrix is as follows:

$$\begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix} = \begin{bmatrix} \frac{A' + B' - C' - D'}{\Delta} & \frac{2 \bullet (A'D' - B'C')}{\Delta} \\ \frac{2}{\Delta} & \frac{-A' + B' - C' + D'}{\Delta} \end{bmatrix}$$

where:

$$\begin{aligned} A' &= A = \cos\left(\frac{2\pi L}{\lambda}\right) & C' &= CZ_o = j\left(\frac{Z}{Z_o}\right) \sin\left(\frac{2\pi L}{\lambda}\right) \\ B' &= \frac{B}{Z_o} = j\left(\frac{Z}{Z_o}\right) \sin\left(\frac{2\pi L}{\lambda}\right) & D' &= D = \cos\left(\frac{2\pi L}{\lambda}\right) \end{aligned}$$

and

$$\Delta = A' + B' + C' + D'$$

Define β as $\frac{2\pi}{\lambda}$. Then $\beta L = \frac{2\pi L}{\lambda}$.

When the mathematical equations are worked out, this leaves:

$$S_{11} = S_{22} = \frac{j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) - \left(\frac{Z_o}{Z}\right) \right]}{2 \cos(\beta L) + j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o}\right) + \left(\frac{Z_o}{Z}\right) \right]}$$

and

$$S_{12} = S_{21} = \frac{2}{2 \cos(\beta L) + j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]}$$

Separating the Expressions

Because we need S-parameters in a real-imaginary format, we need to separate these expressions into their real and imaginary parts by applying their complex conjugate:

$$S_{11} = S_{22} = \frac{j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) - \left(\frac{Z_o}{Z} \right) \right]}{2 \cos(\beta L) + j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]} \bullet \frac{2 \cos(\beta L) - j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]}{2 \cos(\beta L) - j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]}$$

This multiplication yields:

$$Re[S_{11}] = \frac{\sin^2(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right)^2 - \left(\frac{Z_o}{Z} \right)^2 \right]}{4 \cos^2(\beta L) + \sin^2(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]^2} = Re[S_{22}]$$

$$Im[S_{11}] = \frac{2 \sin(\beta L) \cos(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) - \left(\frac{Z_o}{Z} \right) \right]}{4 \cos^2(\beta L) + \sin^2(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]^2} = Im[S_{22}]$$

and

$$S_{12} = S_{21} = \frac{2}{2 \cos(\beta L) + j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]}$$

$$\bullet \frac{2 \cos(\beta L) - j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]}{2 \cos(\beta L) - j \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]}$$

This multiplication yields:

$$Re[S_{21}] = \frac{4 \cos(\beta L)}{4 \cos^2(\beta L) + \sin^2(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]^2 - 2 \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) - \left(\frac{Z_o}{Z} \right) \right]} = Re[S_{12}]$$

$$Im[S_{21}] = \frac{-2 \sin(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) - \left(\frac{Z_o}{Z} \right) \right]}{4 \cos^2(\beta L) + \sin^2(\beta L) \bullet \left[\left(\frac{Z}{Z_o} \right) + \left(\frac{Z_o}{Z} \right) \right]^2} = Im[S_{12}]$$

Note that the expressions all have the same denominator, which makes them easier to code.

Algorithms

Let us go through an algorithm with an example to ensure that it is correct.

Take $Z = 50 \Omega$, $Z_o = 50 \Omega$, and $L = l$. Then $\beta L = 2\pi$. This yields:

$$Re[S_{11}] = 0. \quad (\sin(2\pi) = 0)$$

$$Im[S_{11}] = \frac{2 \sin(2\pi) \cos(2\pi) \bullet \left[\left(\frac{50}{50} \right) - \left(\frac{50}{50} \right) \right]}{4 \cos^2(2\pi) + \sin^2(2\pi) \bullet (1+1)^2} = 0$$

$$Re[S_{21}] = \frac{4 \cos(2\pi)}{4 \cos^2(2\pi) + \sin^2(2\pi) \bullet (1+1)^2} = 1$$

$$Im[S_{21}] = \frac{-2 \sin(2\pi) \bullet (1+1)}{4 \cos^2(2\pi) + \sin^2(2\pi) \bullet (1+1)^2} = 0$$

Another example: $Z = Z_o = 50 \Omega$, $\beta L = \pi/2$ (90°)

$$Re[S_{11}] = \frac{\sin^2\left(\frac{\pi}{2}\right) \bullet (1^2 - 1^2)}{4 \cos^2\left(\frac{\pi}{2}\right) + \sin^2\left(\frac{\pi}{2}\right) \bullet (1 + 1)^2} = 0$$

$$Im[S_{11}] = \frac{2 \sin\left(\frac{\pi}{2}\right) \cos\left(\frac{\pi}{2}\right) \bullet (1 - 1)}{4 \cos^2\left(\frac{\pi}{2}\right) + \sin^2\left(\frac{\pi}{2}\right) \bullet (1 + 1)^2} = 0$$

$$Re[S_{21}] = \frac{4 \cos\left(\frac{\pi}{2}\right)}{4 \cos^2\left(\frac{\pi}{2}\right) + \sin^2\left(\frac{\pi}{2}\right) \bullet (1 + 1)^2} = 0$$

$$Im[S_{21}] = \frac{-2\left(\frac{\pi}{2}\right)(1 + 1)}{4 \cos^2\left(\frac{\pi}{2}\right) + \sin^2\left(\frac{\pi}{2}\right) \bullet (1 + 1)^2} = -1$$

The preceding expressions can be used with any transmission line section as long as b (the propagation constant), Z (the impedance), and L (the length) are known.

Applying a Problem to the Coaxial Cable Section

The impedance of a coaxial cable is defined by:

$$Z = \frac{\eta}{2\pi} \ln\left(\frac{B}{A}\right)$$

where:

Z = impedance

η = characteristic impedance of dielectric = $\sqrt{\frac{\mu_o}{\epsilon}}$

B = outside diameter

A = inside diameter

The propagation constant is defined by:

$$\beta = \frac{2\pi}{\lambda} \text{ where } \lambda = \frac{\text{speed of light in vacuum}}{\sqrt{E_r} \bullet FREQ}$$

Therefore, the required parameters are A , B , L , and E_r .

The U2PB example in U2PB.c is an implementation of the above. The relevant defining

data structures are shown below:

```
static UserParamType
U2PB_parms[] =
{
{"A", REAL_data}, {"B", REAL_data}, {"L", REAL_data},
{"K", REAL_data}
};
static UserElemDef U2PB_ELEMENTS[] =
{
"U2PB", /* modelName */
NUM_EXT_NODES, /* # of external nodes */
siz(U2PB_PARMS), /* # of parameters */
U2PB_PARMS, /* # of parameter structure */
PRE_ANALYSIS_FCN_PTR, /* pre-analysis fcn ptr */
COMPUTE_Y_FCN_PTR, /* Linear model fcn ptr */
COMPUTE_N_FCN_PTR, /* Linear noise model fcn ptr */
POST_ANALYSIS_FCN_PTR, /* post-analysis fcn ptr */
NULL, /* nonlinear structure ptr */
NULL, /* User-defined arb. data structure */
};
```

The a , b , len , and k values are obtained from the circuit through the automatically defined macros:

```
a = A_P;
b = B_P;
len = L_P;
k = K_P;
```

To prevent the program from crashing, some error trapping must be done: a is checked to be positive; b is checked to be greater than a ; and, k is checked to be greater than or equal to 1.

```
if (a <= 0.0 || b <= a || k < 1)
{
(void)sprintf(ErrMsg, "u2pb_y(%s): invalid params: A=%g,
B=%g,K=%g",userInst->tag, a, b, k);
send_error_to_scn(ErrMsg);
return FALSE;
}
```

Calculating Remaining Expressions

The impedance and wave number are then calculated:

```
eta = sqrt(MU0/EPS0/k);
vphase = 1.0 / sqrt(MU0 * EPS0 * k);
betal = omega * len / vphase;
z = eta * log(b / a) / 2.0 / PI;
```

The remaining expressions calculate the S-matrix:

```
zzo= z / ZO;
zoz= ZO / z;
```

```

arg1= zzo - zoz;
arg2= zzo + zoz;
denom = 4.0 * sqr (cos(betal)) + sqr (sin(betal)) * sqr (arg2);
res11 = sqr (sin(betal)) * (sqr(zzo) -sqr (zoz)) / denom;
ims11 = 2.0 * sin(betal) * cos (betal) * arg1 / denom;
res21 = 4.0 * cos(betal) / denom;
ims21 = -2.0 * sin(betal) * arg2 / denom;
S[3].real = S[0].real = res11; (defines S11 .real, S22 .real)
S[3].imag = S[0].imag = ims11; (defines S11 .imag, S22 .imag)
S[2].real = S[1].real = res21; (defines S12 .real, S21 .real)
S[2].imag = S[1].imag = ims21; (defines S12 .imag, S21 .imag)

```

Adding Noise Characteristics

Because the U2PB coaxial section is lossless, it is also noiseless; therefore, the *Noise Analysis Function* check box in the Model Kit Development dialog box is not selected.

The next step is compiling and linking the C-code. See *Building User-Compiled Analog Models* (modbuild).

Creating Nonlinear Circuit Elements

This section describes creating nonlinear circuit elements. Nonlinear user-defined element modeling described in this section is applicable to steady-state analysis only. Refer to *Creating Transient Circuit Elements* (modbuild), for information on modeling the transient response.

Requirements for Creating Nonlinear Elements

A nonlinear circuit element is characterized as follows:

- a linear part
- a nonlinear part
- a bias-dependent small-signal ac part
- a bias-dependent noise part

The first two are mutually exclusive partitions of the element model, while the small-signal part is a combination. All parts can use parameter data of the element item as well as those of any referenced items in the circuit.

The parts are coded as the following function entries in the element's device definition:

- *analyze_lin*
- *analyze_nl*
- *analyze_ac*
- *analyze_ac_n*

Linear Part

The linear part is computed in frequency domain. The user code must compute the branch admittances in the same way as in the linear element case (Refer to *Creating Linear Circuit Elements* (modbuild)). The difference here is in the loading of the circuit nodal admittance matrix, which must be done through the `add_lin_y` function call for each contribution separately. The `analyze_lin` function is called once for every sweep (frequency, power, swept variable) value. The function can be set to `NULL` if the element is completely nonlinear.

Nonlinear Part

The nonlinear part is evaluated on a sample-by-sample basis of time domain pin voltages. The device's `analyze_nl` function must compute the instantaneous nonlinear currents, charges, and their voltage derivatives. Given the user item pin voltages-in the order of external followed by internal-the user-written code computes the nonlinear charges at

each pin and the nonlinear currents out of each pin.

The partial derivatives of these nonlinear quantities with respect to each pin voltage are then computed to formulate conductances and capacitances to load into the circuit Jacobian matrix. In nonlinear analyses, the derivatives influence the rate of convergence, but have no effect on the final steady-state solution. In addition to the instantaneous voltages, delayed pin voltages can be obtained through the `get_delay_v` function.

You may keep any static/intermediate computed data (data invariant over subsequent time samples and iterations) with the particular element item itself. Functions within ADS perform the required time-to-frequency transformations.

AC Part

The ac part linearizes the element model around the dc bias point, and returns the small-signal frequency domain admittance and normalized noise correlation parameters.

The dc bias is determined for the entire flattened circuit, including nonlinear user element items, whose linear and nonlinear parts would be computed as above. Then the device's `analyze_ac` function is called to load the device admittances for all its branches (including internal) into the circuit nodal admittance matrix. If the conductances and capacitances are frequency independent, this will be a combination of the `analyze_lin` and linearized `analyze_nl` functions.

The `analyze_ac_n` function must load the bias-dependent noise current correlation parameters (normalized to `FOUR_K_TO`) using the interface function `add_lin_n` . It is called only if a noise measurement is required in a test bench.

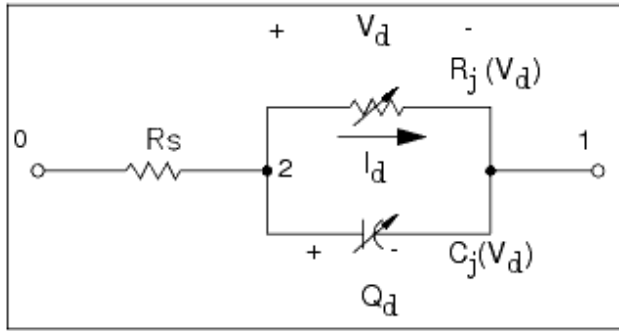
The `UserElemDef` declaration for a nonlinear element has the `compute_y` function automatically set to `NULL` .

User-defined P-N Diode Model

This example shows how to create a nonlinear model of a P-N diode. The result is a set of functions (available in the example `PNDIODE` that provide a simplified model of the ADS `DIODE` element.

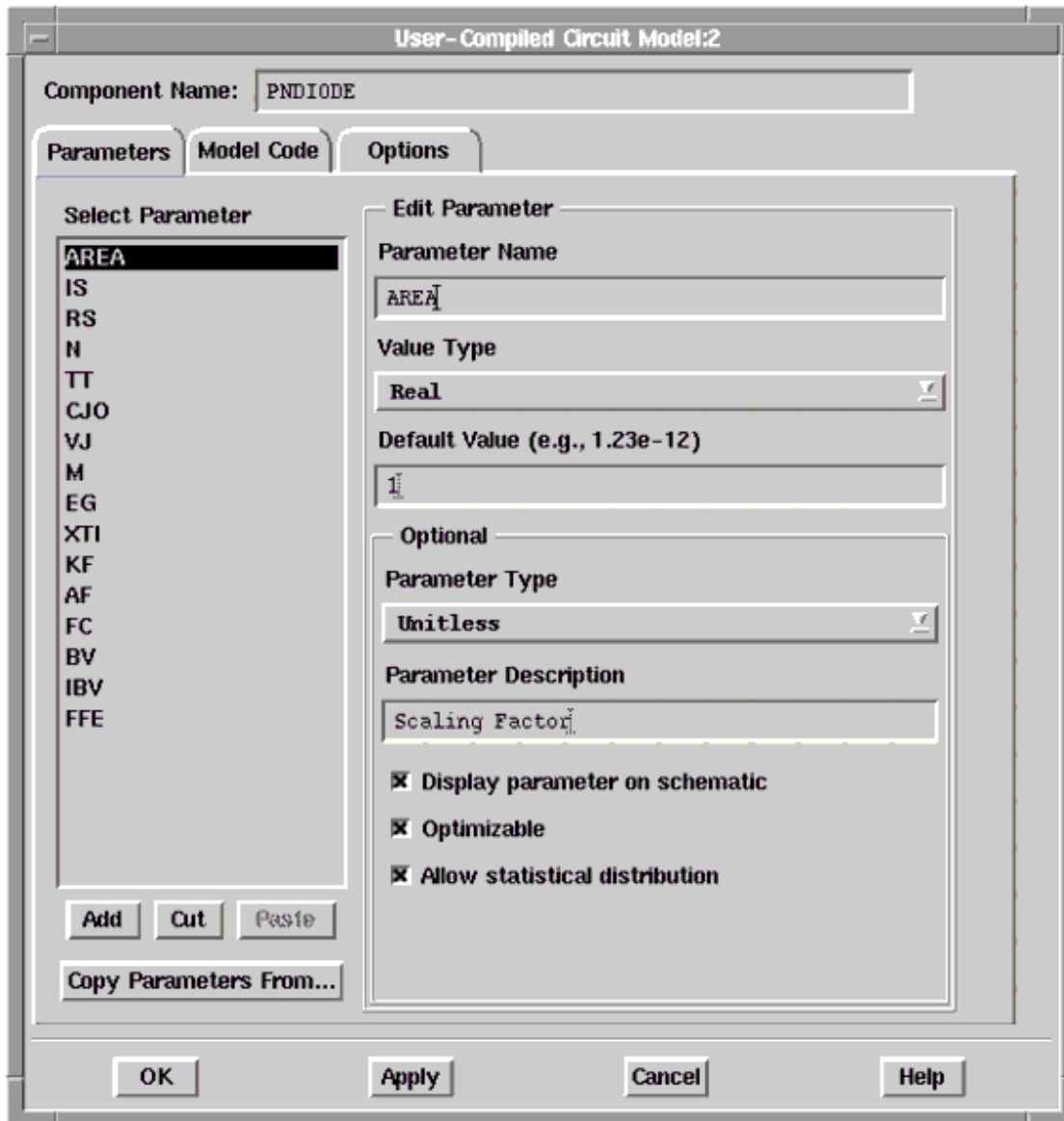
The model shown in the following schematic is used for the `PNDIODE` element.

Figure: PNDIODE element model



Defining a Nonlinear Element

For simplicity, all diode parameters are defined with the element itself in the `UserParamType` array `PNDIODE`, instead of an indirect, shareable model form reference (in Series IV these were referred to as data items). These definitions are entered in the Parameters tab dialog box:



The associated declarations are automatically generated in the PNDIODE_h.c file:

```
static UserParamType
PNDIODE_parms[] =
{
  {"AREA", REAL_data}, {"IS", REAL_data}, {"RS", REAL_data},
  {"N", REAL_data}, {"TT", REAL_data},
  {"CJO", REAL_data}, {"VJ", REAL_data},
  {"M", REAL_data}, {"EG", REAL_data},
  {"XTI", REAL_data}, {"KF", REAL_data},
  {"AF", REAL_data}, {"FC", REAL_data},
  {"BV", REAL_data}, {"IBV", REAL_data},
  {"FFE", REAL_data}
};
```

The associated AEL create_item declarations are also generated in the PNDIODE.ael file.

The three function entries required in a user nonlinear device definition are declared automatically:


```
static boolean analyze_lin(UserInstDef *userInst, double omega);

static boolean analyze_nl(UserInstDef *userInst, double *vPin);

static boolean analyze_ac(UserInstDef *userInst, double *vPin, double omega)
```

The diode has one internal pin between the linear RS and the nonlinear R || C representing the junction. The device definition is (again, automatically-generated):

```
#define ANALYZE_NL_FCN_PTR      analyze_nl
#define ANALYZE_LIN_FCN_PTR    analyze_lin
#define ANALYZE_AC_FCN_PTR     analyze_ac
#define NUM_NONLINEAR_INT_NODES 1
#define ANALYZE_AC_N_FCN_PTR   NULL
static UserNonLinDef
ANALYZE_NL_DEF_PTR =
{
    NUM_NONLINEAR_INT_NODES, /* numIntNodes */
    ANALYZE_LIN_FCN_PTR,    /* analyze_lin() */
    ANALYZE_NL_FCN_PTR,    /* analyze_nl() */
    ANALYZE_AC_FCN_PTR,    /* analyze_ac() */
    NULL,                  /* Nonlin modelDef (user can change) */
    ANALYZE_AC_N_FCN_PTR,  /* analyze_ac_n() */
};
```

The entry for the diode element definition itself is completed, using its parameters and device definition:

```
#define NUM_EXT_NODES          2
#define ANALYZE_NL_DEF_PTR    analyze_nl_def_ptr
#define COMPUTE_Y_FCN_PTR     NULL
#define PRE_ANALYSIS_FCN_PTR  NULL
#define POST_ANALYSIS_FCN_PTR NULL
#define ANALYZE_TR_FCN_PTR    NULL
#define ANALYZE_AC_N_FCN_PTR  NULL
#define COMPUTE_N_FCN_PTR     NULL
#define PNDIODE_PARMS          PNDIODE_parms
#define PNDIODE_ELEMENTS       PNDIODE_elements
static UserElemDef PNDIODE_ELEMENTS[] =
{
    "PNDIODE", /* modelName */
    NUM_EXT_NODES, /* # of external nodes */
    siz(PNDIODE_PARMS), /* # of parameters */
    PNDIODE_PARMS, /* # of parameter structure */
    PRE_ANALYSIS_FCN_PTR, /* pre-analysis fcn ptr */
    COMPUTE_Y_FCN_PTR, /* Linear model fcn ptr */
    COMPUTE_N_FCN_PTR, /* Linear noise model fcn ptr */
    POST_ANALYSIS_FCN_PTR, /* post-analysis fcn ptr */
    &ANALYZE_NL_DEF_PTR, /* nonlinear structure ptr */
    NULL, /* User-defined arb. data structure */
};
```

Implementation of the preceding functions is described next. (Error message reporting, while useful for debugging, is not shown below.) The C macros conveniently centralize parameter indexing:

```
#define AREA_P userInst->pData[0].value.dVal
#define IS_P userInst->pData[1].value.dVal
#define RS_P userInst->pData[2].value.dVal
#define N_P userInst->pData[3].value.dVal
#define TT_P userInst->pData[4].value.dVal
#define CJO_P userInst->pData[5].value.dVal
```

User-Defined Models

```
#define VJ_P  userInst->pData[6].value.dVal
#define M_P   userInst->pData[7].value.dVal
#define EG_P  userInst->pData[8].value.dVal
#define XTI_P userInst->pData[9].value.dVal
#define KF_P  userInst->pData[10].value.dVal
#define AF_P  userInst->pData[11].value.dVal
#define FC_P  userInst->pData[12].value.dVal
#define BV_P  userInst->pData[13].value.dVal
#define IBV_P userInst->pData[14].value.dVal
#define FFE_P userInst->pData[15].value.dVal
```

The linear contribution is just from the series resistor RS between pins 0 and 1. This is coded in the `analyze_lin` function. This function is also called from the small-signal `analyze_ac` function described later.

```
/*-----*/
static boolean add_y_branch(
    UserInstDef *userInst,
    int n1,
    int n2,
    COMPLEX y)
{
    boolean status = TRUE;
    status = add_lin_y(userInst, n1, n1, y) &&
        add_lin_y(userInst, n2, n2, y);
    if (status)
    {
        y.real = -y.real; y.imag = -y.imag;
        status = add_lin_y(userInst, n1, n2, y) &&
            add_lin_y(userInst, n2, n1, y);
    }
    return status;
}
/*-----*/
static boolean analyze_lin (
    UserInstDef *userInst,
    double omega)
{
    boolean status;
    COMPLEX y;
    UserParamData *pData = userInst->pData;
    y.real = y.imag = 0.0;
    if (RS_P > RMIN)
        y.real = AREA_P / RS_P;
    else
        y.real = GMAX;
    status = add_y_branch(userInst, 0, 2, y);
    if (!status)
    {
        (void)sprintf(ErrMsg, "analyze_lin(%s) -> add_lin_y() failed", userInst->tag);
        send_error_to_scn(ErrMsg);
    }
    return status;
}
}
```

The nonlinear device model is coded as a common function (`diode_nl_iq_gc` that follows) so that it can be called from both `analyze_nl` and `analyze_ac` . It computes the nonlinear junction charge, current and their derivatives with respect to the junction voltage.

```
static void diode_nl_iq_gc (
    UserInstDef *userInst, /* Changed from SIV to be consistent w/CUI */
    double *vPin,
    double *id,
    double *qd,
    double *gd,
    double *capd)
{
```

```

double vd, csat, vte, evd, evrev;
double exparg;
double fcpb, xfc, f1, f2, f3;
double czero, arg, sarg, czof2;
UserParamData *pData = userInst->pData;
csat = IS_P * AREA_P;
vte = N_P * VT;
vd = vPin[2] - vPin[1]; /* junction voltage */
/*
 * compute current and derivatives with respect to voltage
 */
if ( vd >= -5.0*vte )
{
    if (vd/vte < 40.0)
    {
        evd = exp(vd/vte);
        *id = csat * (evd - 1.0) + GMIN * vd;
        *gd = csat * evd / vte + GMIN;
    }
    else
    {
        /* linearize the exponential above vd/vte=40 */
        evd = (vd/vte+1.0-40.0)*exp(40.0);
        *id = csat * (evd - 1.0) + GMIN * vd;
        *gd = csat * exp(40.0) / vte + GMIN;
    }
}
else
{
    *id = -csat + GMIN * vd;
    *gd = -csat / vd + GMIN;
    if ( BV_P != 0.0 && vd <= (-BV_P+50.0*VT) )
    {
        exparg = ( -(BV_P+vd)/VT < 40.0 ) ? -(BV_P+vd)/VT : 40.0;
        evrev = exp(exparg);
        *id -= csat * evrev;
        *gd += csat * evrev / VT;
    }
}
/*
 * charge storage elements
 */
fcpb = FC_P * VJ_P;
czero = CJO_P * AREA_P;
if (vd < fcpb)
{
    arg = 1.0 - vd / VJ_P;
    sarg = exp(-M_P * log(arg));
    *qd = TT_P * (*id) + VJ_P * czero * (1.0 - arg * sarg)
    / (1.0 - M_P);
    *capd = TT_P * (*gd) + czero * sarg;
}
else
{
    xfc = log(1.0 - FC_P);
    /* f1 = vj*(1.0-(1.0-fc)^(1.0-m))/(1.0-m) */
    f1 = VJ_P * (1.0-exp((1.0-M_P)*xfc)) / (1.0-M_P);
    /* f2 = (1.0-fc)^(1.0+m) */
    f2 = exp((1.0+M_P)*xfc);
    /* f3=1.0-fc*(1.0+m) */
    f3 = 1.0 - FC_P * (1.0+M_P);
    czof2 = czero / f2;
    *qd = TT_P * (*id) + czero * f1 + czof2 * (f3 * (vd - fcpb) +
    (M_P / (VJ_P + VJ_P)) * (vd * vd - fcpb * fcpb));
    *capd = TT_P * (*gd) + czof2 * (f3 + M_P * vd / VJ_P);
}
} /* diode_nl_iq_gc() */

```

The following equation is used for the diode current I_d in the forward bias mode.
For $V_d \leq 40 \cdot V_{te}$:

$$I_d(V_d) = I_s \left(e^{\frac{V_d}{V_{te}}} - 1 \right) + 10^{-12} \cdot V_d$$

The derivative of the diode current with respect to the junction voltage is:

$$\frac{dI_d}{dV_d} = \frac{I_s \cdot e^{\frac{V_d}{V_{te}}}}{V_{te}} + 10^{-12}$$

For $V_d > 40 \cdot V_{te}$, the exponential is linearized to prevent numerical overflow:

$$I_d(V_d) = I_s[(V_d/V_{te} + 1 - 40)e^{40} - 1] + 10^{-12} \cdot V_d$$

$$\frac{dI_d}{dV_d} = \frac{I_s \cdot e^{40}}{V_{te}} + 10^{-12}$$

For the case $V_d < FC \cdot VJ$, the total charge and large-signal incremental capacitance expressions are:

$$Q_d(V_d) = TT \cdot I_d + \frac{VJ \cdot CJO}{(1-M)} \cdot \left[1 - \left(1 - \frac{V_d}{VJ} \right)^{1-M} \right]$$

$$C_d(V_d) = \frac{dQ_d}{dV_d} = TT \cdot \frac{dI_d}{dV_d} + CJO \left(1 - \frac{V_d}{VJ} \right)^{-M}$$

The `analyze_nl` function that follows loads the nonlinear currents, charges at each pin, and the nonlinear conductances, capacitances for each branch. Note that each G, C component has four Jacobian matrix contributions, two diagonals and two off-diagonals.

```
static boolean analyze_nl (
    UserInstDef *userInst,
    double *vPin)
{
    double id, gd; /* current, conductance */
    double qd, capd; /* charge, capacitance */
    boolean status;
    char *pMsg = NULL;
    diode_nl_iq_gc(userInst, vPin, &id, &qd, &gd, &capd);
    /*
     * load nonlinear pin currents out of each terminal and
     * nonlinear charges at each terminal.
     */
    status = add_nl_iq(userInst, 1, -id, -qd) &&
             add_nl_iq(userInst, 2, id, qd);
    if (!status)
    {
        pMsg = "add_nl_iq()";
        goto END;
    }
    /* Add nonlinear conductance, capacitance
     * 0      1      2
     * 0
     * 1      Y      Y
     * 2      Y      Y
    */
}
```

```

*/
status = add_nl_gc(userInst, 1, 1, gd, capd) &&
        add_nl_gc(userInst, 1, 2, -gd, -capd) &&
        add_nl_gc(userInst, 2, 1, -gd, -capd) &&
        add_nl_gc(userInst, 2, 2, gd, capd);
if (!status)
    pMsg = "add_nl_gc()";
END:
if (pMsg)
{
    (void)sprintf(ErrMsg, "Error: PNDIODE: analyze_nl(%s) -> %s", userInst->tag, pMsg);
    send_error_to_scn(ErrMsg);
}
return status;
} /* analyze_nl() */

```

The `analyze_ac` function that follows characterizes the PNDIODE's bias-dependent small-signal ac behavior. It calls `analyze_lin` to load the linear part, then loads the linearized admittances obtained from the nonlinear junction conductance and capacitance at the DC bias point.

```

static boolean analyze_ac (
    UserInstDef *userInst,
    double *vPin,
    double omega)
{
    COMPLEX y;
    double id, gd; /* current, conductance */
    double qd, capd; /* charge, capacitance */
    boolean status;
    /*
     * Add linearized conductance, susceptance
     *      0      1      2
     *      0  G      G
     *      1      Y  Y
     *      2  G  Y  Y
     */
    if (!analyze_lin(userInst, omega))
        return pw;
    diode_nl_iq_gc(userInst, vPin, &id, &qd, &gd, &capd);
    y.real = gd; y.imag = omega * capd;
    status = add_y_branch(userInst, 1, 2, y);
    if (!status)
    {
        (void)sprintf(ErrMsg, "Error: PNDIODE: analyze_ac(%s) -> add_lin_y", userInst->tag);
        send_error_to_scn(ErrMsg);
    }
    return status;
} /* analyze_ac() */

```

The `analyze_ac_n` function that follows models the PNDIODE's noise behavior in a linear analysis. It loads the device's thermal noise, and its bias-dependent shot and flicker noise contributions through the interface primitive function `add_lin_n`. The static function `add_n_branch` loads a branch contribution symmetrically into the circuit's indefinite noise-current correlation matrix.

```

/*-----*/
static boolean add_n_branch(
    UserInstDef *userInst,
    int n1,
    int n2,
    COMPLEX iNcorr)
{
    boolean status = TRUE;
    status = add_lin_n(userInst, n1, n1, iNcorr) &&

```

User-Defined Models

```
add_lin_n(userInst, n2, n2, iNcorr);
if (status)
{
    iNcorr.real = -iNcorr.real; iNcorr.imag = -iNcorr.imag;
    status = add_lin_n(userInst, n1, n2, iNcorr) &&
    add_lin_n(userInst, n2, n1, iNcorr);
}
return status;
}
/*-----*/
static boolean analyze_ac_n (
    UserInstDef *userInst,
    double *vPin,
    double omega)
{
    double id, gd; /* current, conductance */
    double qd, capd; /* charge, capacitance */
    boolean status;
    COMPLEX thermal, dNoise; /* noise-current correlation admittance */
    double kf, gs, tempScale;
    char *pMsg = NULL;
    UserParamData *pData = userInst->pData;
    diode_n1_iq_gc(userInst, vPin, &id, &qd, &gd, &capd);
    tempScale = DEV_TEMP / NOISE_REF_TEMP;
    dNoise.imag = thermal.imag = 0.0;
    if (RS_P > RMIN)
        gs = AREA_P / RS_P;
    else
        gs = GMAX;
    thermal.real = tempScale * gs;
    id = fabs(id);
    kf = fabs(KF_P);
    /* shot noise */
    dNoise.real = 2.0 * CHARGE * id;
    /* flicker noise */
    if (id > 0.0 && omega > 0.0 && kf > 0.0)
        dNoise.real += kf * pow(id, AF_P) * pow(omega/TWOPI, -FFE_P);
    dNoise.real /= FOUR_K_TO;
    status = add_n_branch(userInst, 0, 2, thermal) &&
    add_n_branch(userInst, 1, 2, dNoise);
    if (!status)
        pMsg = "add_lin_n()";
    if (pMsg)
    {
        (void)sprintf(ErrMsg, "Error: analyze_ac_n(%s) -> %s", userInst->tag, pMsg);
        send_error_to_scn(ErrMsg);
    }
    return status;
} /* analyze_ac_n() */
```

The next step is compiling and linking the code. Refer to *Creating the Code and Compiling the Model* (modbuild).

Referencing Data Items

Refer to *Referencing Data Items* (modbuild).

Displaying Error/Warning Messages

Refer to *Displaying Error/Warning Messages* (modbuild).

Creating Transient Circuit Elements

This section describes the steps necessary for creating transient circuit elements.

Requirements for Creating Transient Elements

A transient model can be created for either a linear or nonlinear element. A transient element can have additional nodes internal to the element, as specified by the value of `numIntNodes` (set in the Code Options dialog field No. of internal nodes under the Transient Function check box. If a nonlinear model (using `UserNonLinDef`) is defined for the element, the `numIntNodes` in that structure must match this definition.

Special routines are available to simplify the use of ideal resistors, capacitors, inductors, and transmission lines within a transient element. For all but resistors, the number of these elements must be predefined using `numCaps`, `numInds`, and `numTlines` so the circuit simulator engine can perform the appropriate allocations. These values are entered in the appropriate fields in the Code Options dialog box.

The time-domain response is evaluated from the instantaneous pin voltages. The device's `analyze_tr` function must compute the instantaneous nonlinear currents, charges and their voltage derivatives. Given the instantaneous user item pin voltages (external first, followed by internal, starting at zero), the user-defined code computes the nonlinear charges at each pin and the nonlinear currents out of each pin. The partial derivatives of these nonlinear quantities with respect to each pin voltage are then computed to formulate conductances and capacitances to load into the circuit Jacobian matrix.

If P is the total number of pin voltages, formulate nonlinear current and charge at each pin n as follows:

$$r_n(t) = f(v_0(t), v_1(t), \dots, v_{P-1}(t))$$

where r_n is the pin current or charge response.

These responses and their voltage derivatives (nonlinear conductances and capacitances) must be computed and loaded using the `add_tr_iq` and `add_tr_gc` functions, respectively.

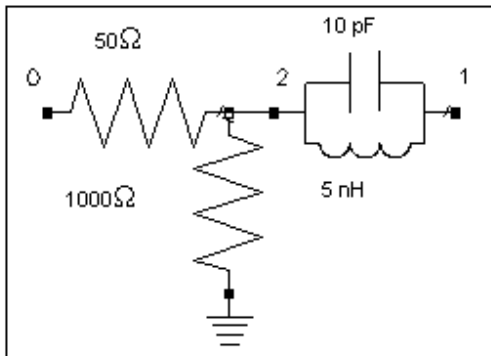
Using Resistors, Capacitors, and Inductors

Special routines are available to simplify the use of ideal resistors, capacitors, inductors, and transmission lines within a transient element. These routines can be called from within the user-written `analyze_tr` function. For all but resistors, the number of these elements must be predefined using `numCaps`, `numInds` and `numTlines`, so the circuit simulator engine can perform the appropriate allocations.

If the requested number of elements is not used, an error message is generated and the analysis fails. The following code example could be used within an `analyze_tr` function to implement a transient model for the element model shown in the following figure. Naturally, values for these components could be calculated from the `UserInstDef` parameters that are passed into the function.

```
boolean example1_tr (UserInstDef *pInst, double *vPin)
{
    boolean status;
    status = add_tr_resistor(pInst, 0, 2, 50.0) &&
    add_tr_resistor(pInst, 2, GND, 1000.0) &&
    add_tr_capacitor(pInst, 2, 1, 10.0e-12) &&
    add_tr_inductor(pInst, 2, 1, 5.0e-9);
    return status;
}
```

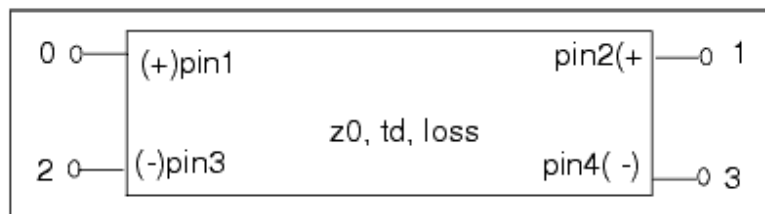
Figure1: Element model for transient analysis



Using Transmission Lines

The `fix_tr` function is called just before transient analysis begins. Its only purpose is to set up ideal transmission lines for the user. Using the `add_tr_tline` function, transmission line pins and physical constants are defined. All four terminals of the transmission line are available to the user (See the following figure). Once the transmission line is defined here, time-domain analysis of it is performed automatically without any further action by the user in the `analyze_tr` function.

Figure2: Four transmission line terminals



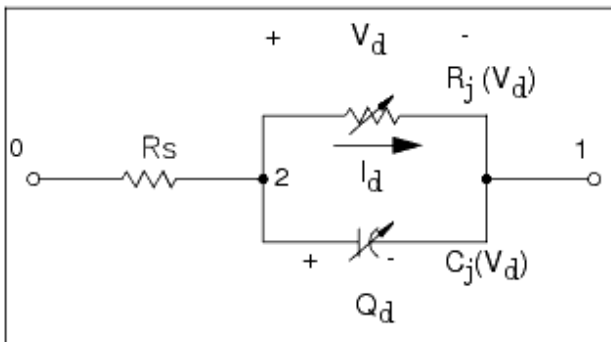
The following code sample places a lossless 50Ω transmission line with a 100 psec delay between pins 0 and 1. An `analyze_tr` function is still required, even though it doesn't do anything-it should simply return TRUE.

```
boolean example2_fix (UserInstDef *pInst)
{
    boolean status;
    status = add_tr_tline(pInst, 0, 1, GND, GND, 50.0, 100.0e-12, 1.0);
    return status;
};
boolean example2_tr (UserInstDef *pInst, DOUBLE *vPin)
{
    return TRUE;
}
```

User-defined P-N Diode Model

This section shows how to extend the P-N diode example created in *Creating Nonlinear Circuit Elements* (modbuild) for use in a transient model. Only the new code and modifications required to extend this element to a transient model are listed below. The code for this model is available in the example `PNDIODE`. The model shown in the following figure is used for the `PNDIODE` element.

Figure3: PNDIODE element model



Defining the Transient Device

A prototype for the transient analysis function is required and is automatically generated in the `PNDIODE_h.c` file when the Transient Function check box is selected in the Code Options dialog box.

```
#define ANALYZE_TR_FCN_PTR    analyze_tr
static boolean analyze_tr(UserInstDef *userInst, double *vPin);
```

A `UserTranDef` structure is defined for transient.

```
#define ANALYZE_TR_DEF_PTR    analyze_tr_def_ptr
static UserTranDef static UserTranDef
ANALYZE_TR_DEF_PTR =
{
NUM_TRANSIENT_INT_NODES,    /* numIntNodes */
NUM_TRANSIENT_CAPS,        /* numCaps */
NUM_TRANSIENT_INDS,        /* numInds */
NUM_TRANSIENT_TLNS,        /* numTlins */
USE_CONVOLUTION,           /* useConvolution */
ANALYZE_TR_FCN_PTR,        /* analyze_tr */
FIX_TR,                    /* fix_tr */
};
```

In the UserElemDef, a pointer to the DIODE_TR structure is added at the end via a macro.

```
#define ANALYZE_TR_DEF_PTR    analyze_tr_def_ptr
static UserElemDef PNDIODE_ELEMENTS[] =
{
"PNDIODE",    /* modelName */
..
&ANALYZE_NL_DEF_PTR,    /* nonlinear structure ptr */
NULL,                  /* User-defined arb. data structure */
&ANALYZE_TR_DEF_PTR,    /* transient fcn ptr */
}
```

Transient Analysis Function

The analysis routine `diode_nl_iq_gc` that was written for the nonlinear model (Refer to *Creating Nonlinear Circuit Elements* (modbuild)) can also be used for the transient model. Add to this the contribution of the series resistance and the model is complete. The `analyze_tr` function that follows calls the `diode_nl_iq_gc` function for the nonlinear contribution and loads them into the matrix, and then uses `add_tr_resistor` to include the contribution of the series resistance.

```
static boolean analyze_tr(
UserInstDef *userInst,
double      *vPin)
{
UserParamData *pData = userInst->pData;
char *pMsg = NULL;
boolean status;
double id, qd, gd, capd, rs;
/* compute the nonlinear portion */
diode_nl_iq_gc(userInst, vPin, &id, &qd, &gd, &capd);
status = add_tr_iq(userInst, 2, id, qd) &&
add_tr_iq(userInst, 1, -id, -qd);
if (status == FALSE) goto END;
status = add_tr_gc(userInst, 2, 2, gd, capd) &&
add_tr_gc(userInst, 2, 1, -gd, -capd) &&
add_tr_gc(userInst, 1, 2, -gd, -capd) &&
add_tr_gc(userInst, 1, 1, gd, capd);
if (status == FALSE) goto END;
/* series resistance */
if (AREA_P > 0.0)
rs = RS_P / AREA_P;
else
rs = 0.0;
status = add_tr_resistor(userInst, 0, 2, rs);
END:
if (pMsg)
{
```

User-Defined Models

```
(void)sprintf(ErrMsg, "Error: PNDIODE: analyze_tr(%s) -> %s", userInst->tag, pMsg);
send_info_to_sch(ErrMsg);
}
return status;
} /* analyze_tr() */
```

The next step is compiling and linking the code. Refer to *Building User-Compiled Analog Models* (modbuild).

Referencing Data Items

Refer to *Referencing Data Items* (modbuild).

Displaying Error/Warning Messages

Refer to *Displaying Error/Warning Messages* (modbuild).

Custom Modeling with Symbolically-Defined Devices

This section presents a powerful capability of Advanced Design System: the ability to create a user-defined nonlinear component which can simulate both the large-signal and small-signal behavior of a nonlinear device, without the use of source code.

The *symbolically-defined device* (SDD) is an equation-based component that enables you to quickly and easily define custom, non-linear components. These components are multi-port devices that can be modeled directly on a schematic. You define an SDD by specifying equations that relate port currents, port voltages, and their derivatives. Equations can also reference the current flowing in another device. Once a model is defined, it can be used with any circuit simulator in Advanced Design System. Derivatives are automatically calculated during the simulation.

Before the SDD, the techniques that were available for modeling nonlinear devices were either limited or cumbersome. One technique was to model the device equations using discrete components—usually resistors, capacitors, inductors, and controlled sources. Since most simulators restrict these devices to be linear, this approach could be used to model only the small-signal (AC) behavior of the nonlinear device, and you could not achieve an accurate DC simulation or harmonic balance simulation. A second approach would be to use measured data, typically S-parameters, to model the device, but this approach, too, modeled only small-signal behavior.

The only technique previously available to develop a model that simulated both the large-signal and small-signal behavior of a nonlinear device required writing source code, which was a lengthy task. For example, a typical BJT model would require over 4500 lines of code, and could take an experienced engineer well over a month to write and debug. There also is the requirement that the simulator be linked to your compiled code.

By comparison, the SDD offers a simple, fast way to develop and modify complex models. Equations can be modified easily, and simulation results can be compared to measured data within Advanced Design System.

The SDD can also model high-level circuit blocks such as mixers or amplifiers. By using a single, high-level component instead of a subcircuit of low-level devices, simulations run more quickly. If second- and third-order effects of low-level subcircuits need to be analyzed, the SDD can be modified to develop a more comprehensive implementation of the circuit.

The examples in this section start with a simple nonlinear resistor, then more complex devices, like the Gummel-Poon charge-storage model of the bipolar junction transistor, are described. With the techniques used to develop these models, you can develop your own, custom, nonlinear components.

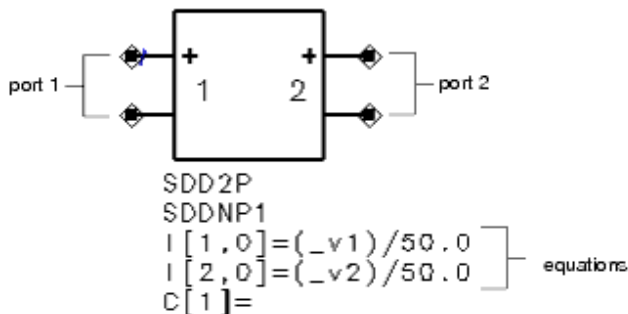
This section has the following sections:

- [Writing SDD Equations](#) explains how to write the equations that define an SDD.
- [Adding an SDD to a Schematic](#) describes how to add an SDD to a schematic and enter equations.
- [SDD Examples](#) show how to use SDDs to define a wide range of nonlinear circuit components.
- [Modified Nodal Analysis](#) is a discussion of modified nodal analysis and branch equations.
- [Error Messages](#) lists SDD error messages and their meaning.

Detailed knowledge of microwave and RF circuit theory and of building and analyzing circuits using Advanced Design System is assumed.

Writing SDD Equations

The symbolically-defined device is represented on the circuit schematic as an n -port device, with up to 10 ports. The equations that specify the voltage and current of a port are defined as functions of other voltages and currents. An example of a 2-port SDD is shown here.



The schematic symbol for a two-port SDD

Port Variables

For each port on the SDD, there are voltage and current *port variables*. A variable begins with an underscore, followed by v (for voltage) or i (for current), and the port number. For example, current and voltage variables for port one are $_i1$ and $_v1$, respectively. You can rename variables to better suit the device being modeled. In text, v_n and i_n are used to refer to $_vn$ and $_in$.

By convention, a positive port current flows into the terminal marked +.

Defining Constitutive Relationships with Equations

A well-defined n port is described by n equations, called *constitutive relationships*, that relate the n port currents and the n port voltages. For linear devices, the constitutive relationships are often specified in the frequency domain (for example, as admittances), but since the SDD is used to model nonlinear devices, its constitutive relationships are specified in the time domain.

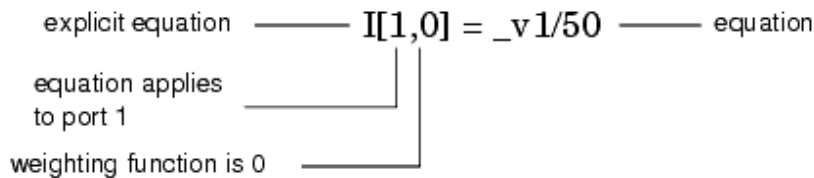
The constitutive relationships may be specified in either *explicit* or *implicit* representations.

Explicit Representation

With the explicit representation, the current at port k is specified as a function of port voltages:

$$i_k = f(v_1, v_2, \dots, v_n)$$

An example of an explicit equation is:



In this example, the current at port 1 is calculated by dividing the voltage at port 1 by 50.



Note

Each port of the SDD must have at least one equation. For an unused port n , apply the equation $I[n,0] = 0.0$ (an open circuit) to the unused port.



Note

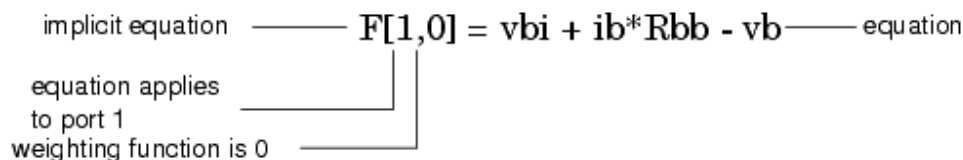
Although not often utilized in standard circuit models, the explicit equation defining the i_k port current can actually be a function of any of the port voltages *and* any of the other port currents for ports that are defined with implicit equations. Since port k is being defined with an explicit equation, the i_k port variable is not available and so cannot be used to implicitly define i_k .

Implicit Representation

The implicit representation uses an implicit relationship between any of the port currents and any of the port voltages:

$$f_k(v_1, v_2, \dots, v_n, i_1, i_2, \dots, i_n) = 0$$

An example of an implicit equation is:



This equation is part of the Gummel-Poon example.

If you want to use the current variable (i_n) of a port in another equation, you must define the port using an implicit equation.

A procedure for how to enter equations is in the section [Adding an SDD to a Schematic](#).

Explicit Versus Implicit Representations

The explicit representation is a voltage-controlled representation and can implement only voltage-controlled expressions. The implicit representation has no such restriction. It can model equations that are voltage-controlled, current-controlled, or use some other control.

Although implicit equations have no restrictions, explicit equations are more "natural" and more efficient. The explicit representation is more natural simply because many models are expressed in the voltage-controlled form $i = f(v)$. The corresponding implicit equation is $i - f(v) = 0$, which is less intuitive.

Explicit equations use standard nodal analysis, that is, the sum of the currents entering and exiting a node equal zero. Implicit equations use modified nodal analysis, which adds a branch equation and makes i_k available as a variable. For more information on modified nodal analysis, refer to the section [Modified Nodal Analysis](#).

The explicit representation is more efficient during a simulation because it is a voltage-controlled representation and, therefore, does not create any new variables in the modified nodal equations. With implicit equations, for every port that uses an implicit representation, the port current is appended to the list of branch currents and the port equation is appended to the modified nodal analysis equations. The result is a larger system of equations with a larger number of unknowns (for a discussion of modified nodal analysis and branch equations, see the section [Modified Nodal Analysis](#)).

In general, you should use the implicit representation only when the explicit representation is insufficient. For example, for a given port n , the port current variable i_n can be used in other equations only if port n is defined with an implicit equation.

Continuity

Many of the circuit-solving algorithms used by the simulator are based on the Newton-Raphson algorithm. Consequently, constitutive relationships should conform to the following:

- The functions must be continuous with respect to v and i .
- Ideally, the functions should be differentiable with respect to v and i , but it is not required.
- It is desirable if the derivatives are continuous with respect to v and i , but this is not necessary, for example, a step discontinuity in the derivative is often acceptable.

An example where these considerations are important is piecewise-defined devices where the constitutive relationship changes depending on the region of operation. The constitutive relationships should be carefully pieced together to ensure continuous derivatives at the region boundaries. An example is given in [Full Model Diode, with Capacitance and Resistance](#).

Although continuous derivatives are not required, if a constitutive relationship does not have continuous derivatives, the simulator may have trouble converging, even at low power levels. If you are having convergence problems with an SDD, the continuity of derivatives is the first thing to check.

Weighting Functions

A *weighting function* is a frequency-dependent expression used to scale the spectrum of a port current. Weighting functions are evaluated in the frequency domain.

There are two predefined weighting functions. Weighting function 0 is defined to be identically one. It is used when no weighting is desired. Weighting function 1 is defined as $j\omega$ and is used when a time derivative is desired.

You can define other weighting functions, starting with the number 2. Weighting functions must be defined in the frequency domain. Weighting functions can, for example, correspond to time delay or to a low-pass or high-pass filter. An example of a time delay weighting function is:

$$e^{-j \times \omega \times delay}$$

Be aware that the SDD will be evaluated at DC, so a user-defined weighting function should be well behaved at $j\omega=0$. For example, you might want to use a weighting function of $1/j\omega$ to perform time integration, but this will cause a divide-by-zero error at DC.

For information on how to enter weighting functions as part of an SDD definition, refer to [Defining a Weighting Function](#).

Weighting Function Example

To understand how the weighting functions are used, this example outlines the steps taken to evaluate the port current of an SDD during a harmonic balance simulation.

For simplicity, consider a one-port SDD with an explicit representation for port one:

$$I[1,1] = f(v_1)$$

where f is some nonlinear function.

During a harmonic balance simulation, the simulator supplies the SDD with the spectrum $V_1(\omega)$ of the port voltage and asks the SDD for the spectrum $I_1(\omega)$ of the corresponding port current. To evaluate the current, the SDD performs four steps:

$$V_1(\omega) \Rightarrow v_1(t) \Rightarrow \hat{i}_1(t) \Rightarrow \hat{I}_1(\omega) \Rightarrow I_1(\omega)$$

1. Perform an inverse Fourier transform on the voltage spectrum $V_1(\omega)$ to obtain a (sampled) time waveform $v_1(t)$.
2. Evaluate the nonlinearity f point by point along the time waveform. The result is the (sampled) time waveform $\hat{i}_1(t)$.
3. Perform a Fourier transform on the time waveform to obtain its spectrum $\hat{I}_1(\omega)$.
4. Scale the components of this spectrum using the weighting function to obtain the spectrum $I_1(\omega)$ of the port current.


Note

The nonlinearity is evaluated in the time domain. The weighting function is evaluated in the frequency domain.

Since multiplication by $j\omega$ in the frequency domain is equivalent to time differentiation in the time domain, in this example, the current is:

$$i_1(t) = \frac{d}{dt} f(v_1(t))$$

You will see this result used in [Nonlinear Capacitors](#) and [Nonlinear Inductors](#), where the weighting function 1 is used to implement nonlinear capacitors and inductors.

Controlling Currents

Not only can the equations for an SDD be written in terms of its own port voltages and currents, an SDD can also be set up to reference the current flowing in another device. The devices that can be referenced are limited to either voltage sources or current probes in the same network. For instructions on how to define a controlling current, refer to the section [Defining a Controlling Current](#). An example appears in [Controlling Current, Instantaneous Power](#).

Specifying More than One Equation for a Port

It is possible to specify more than one expression for a port, but they must be either all implicit or all explicit expressions. And, each port must have at least one equation. When

more than one expression is given for a port, the SDD calculates a separate spectrum for each expression. Each spectrum is weighted by the weighting function specified for that expression. The SDD then sums up the individual spectra to get the final spectrum. Explicit and implicit examples follow.

Explicit Cases

The two SDD equations

$$I[1,0] = f1_v1$$

$$I[1,0] = f2_v1$$

and

$$I[1,0] = f1_v1 + f2_v1$$

are equivalent and implement

$$i_1 = f_1(v_1) + f_2(v_1)$$

The SDD equations

$$I[1,0] = f1_v1$$

$$I[1,1] = f2_v1$$

implement

$$i_1 = f_1(v_1) + \frac{d}{dt} f_2(v_1)$$

Implicit Cases

The two SDD equations

$$F[1,0] = f1_v1, _i1$$

$$F[1,0] = f2_v1, _i1$$

and

$$F[1,0] = f1_v1, i1 + f2_v1, _i1$$

are equivalent and implement

$$f_1(v_1, i_1) + f_2(v_1, i_1) = 0.$$

In the case of an implicit representation, if there is only one expression for a port or, equivalently, more than one expression for a port but all the expressions use the same weighting function, do not use a weighting function other than 0. To see this, assume that in the previous example the weighting function is not weighting function number 0 but is the user-defined function $H(\omega)$. Then in the frequency domain, the implicit equation becomes

$$H(\omega)F_1(V_1(\omega), I_1(\omega)) + H(\omega)F_2(V_1(\omega), I_1(\omega)) = 0$$

which is equivalent to

$$F_1(V_1(\omega), I_1(\omega)) + F_2(V_1(\omega), I_1(\omega)) = 0$$

Here, upper-case letters are used to indicate frequency-domain values, and this assumes that the weighting function does not evaluate to zero at a frequency of interest.

You would want to use a weighting function other than 0 with an implicit representation when two or more expressions are used for a port and different weighting functions are used by the expressions. For example, the SDD equations in this example:

$$\begin{aligned} F[1,0] &= f1(_v1, _i1) \\ F[1,1] &= f2(_v1, _i1) \end{aligned}$$

implement

$$f_1(v_1, i_1) + \frac{d}{dt} f_2(v_1, i_1) = 0$$

Using an SDD to Generate Noise

An SDD generates noise in all four types of noise analysis: linear (AC and S-parameter), harmonic balance (mixer and phase noise), transient noise, and Circuit Envelope noise. If you want to add 1/f noise to a current source, consider using a standard current noise source and set its value with an equation so it is a function of frequency:

$$I_n = 1e-12 + 1e-6/(freq+1)$$

In the denominator, the 1 is added so that the equation is not divided by zero when $freq=0$.

Summary

- The SDD is an n -port device.
- For port n , the voltage is denoted $_vn$. The current is denoted $_in$. Positive current flows into the terminal marked +.
- The *explicit representation* is useful for voltage-controlled nonlinearities:
 $i = f(v)$

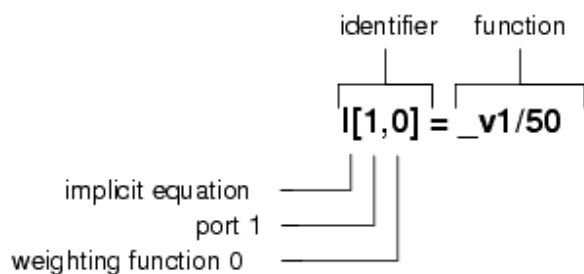
- The *implicit representation* is useful for the general nonlinearity:
 $f(i,v) = 0$
- *Weighting functions* are used to give a frequency weighting to a spectrum. Weighting function number 0 corresponds to no (that is, unity) weighting. Weighting function number 1 corresponds to $j\omega$ and is used to implement a time derivative.
- SDD equations can reference the current flowing in voltage sources or current probes in the same network.
- When more than one expression is given for a port, each expression is evaluated, converted into a spectrum, and weighted separately from the others. The resulting spectra are added together to get the final spectrum.
- An SDD generates noise in all four types of noise analysis.

Adding an SDD to a Schematic

SDDs can be added to a schematic in the same way as other components are added and connected to a circuit. This section describes the mechanics of adding an SDD component to a schematic and defining it.

To add an SDD:

1. From the Component Palette List, choose **Eqn-based Nonlinear**.
2. Select the SDD with the desired number of ports, add it to the schematic, and return to select mode.
3. Double-click the SDD symbol to edit the component.
4. The equations that define the SDD are entered as parameters in the Select Parameters list. The left side of an equation identifies the type of equation, the port it is applied to, and the weighting function. Select the equation you want to edit. (Note the buttons below the list to add, cut, and paste equations as necessary.)



5. Under Parameter Entry Mode, specify the type of equation: **implicit**, or **explicit**. For more information on the types of equations, refer to the section [Defining Constitutive Relationships with Equations](#).
6. In the Port field, enter the number of the port that you want the equation to apply to.
7. In the Weight field, enter the weighting function that you want to use. Predefined weighting functions are 0 (the equation is multiplied by 1) and 1 (the equation is multiplied by $j\omega$). For more information on weighting functions, refer to the section [Weighting Functions](#). For information on the procedure for adding a different weighting function to an SDD, refer to the section [Defining a Weighting Function](#).
8. In the Formula field, enter the equation. For long equations, click **More** for a larger entry area.
9. Click **Apply** to update the equation.
10. Add and edit other equations for other ports as desired.

11. Click **OK** to accept the changes and dismiss the dialog box.

Defining a Controlling Current

The equations for an SDD can be written in terms of the current flowing in another device. For example, you can use the current flowing through a voltage source as part of an SDD equation. You can specify only the current through devices that are either voltage sources or current probes as control currents, and they must be in the same network as the SDD. To specify a current as a control current, you enter the instance name of the device in the $C[]$ parameter of the SDD. For example, to use the current flowing through a voltage source called *SRC1*, you would set the current parameter $C[1]$ to *SRC1*. The SDD equations use the variable $_c1$ to refer to this current.

To define a controlling current:

1. Double-click the SDD component to open the Edit Component dialog box.
2. Select $C[1]=$ in the Select Parameters list.
3. Choose *String and Reference* as the parameter entry mode; *File based* should not be used. In the $C[\text{Repeated}]$ field, type the instance name of the device.
An example of a parameter definition is shown here.

$C[1] = Vdds$

control current	_____
parameter	_____
instance	_____
name	_____

4. To add another controlling current, select $C[1]$ and click **Add**. The parameter $C[2]$ appears in the parameter list. You can define this parameter for another current.
5. Click **Apply** to update the SDD definition.
6. To use the controlling current in an equation, type $_c_n$ in your SDD equation, for example, $_v2 + _v1 * _c1$.
7. Click **OK** to accept the changes and dismiss the dialog box.

Defining a Weighting Function

A *weighting function* is a frequency-dependent expression that is used to scale the spectrum of a port current. Weighting functions are evaluated in the frequency domain. Predefined weighting functions are 0 (the equation is multiplied by 1) and 1 (the equation is multiplied by $j\omega$). You can define your own weighting functions.

To define a weighting function:

1. Double-click the SDD component to open the Edit Component dialog box.
2. Select any equation in the Select Parameters list.
3. Click **Add**. The new equation is automatically selected.
4. From the Parameter Entry Mode list, choose **Weighting**. Note that an **H** appears on the left side of the equation to denote it is a weighting function.
5. In the Weight field, enter a value greater than 1. Each weighting function must have

- a unique value.
6. In the Formula field, enter the weighting function.
 7. Click **Apply** to update the SDD definition.
 8. Click **OK** to accept the changes and dismiss the dialog box.

SDD Examples

This section offers the following detailed examples that show how to use symbolically-defined devices to define a wide range of nonlinear circuit components. The examples include:

- [Nonlinear Resistor](#)
- [Ideal Amplifier Block](#)
- [Ideal Mixer](#)
- [Nonlinear Capacitors](#)
- [Full Model Diode, with Capacitance and Resistance](#)
- [Nonlinear Inductors](#)
- [Controlling Current, Instantaneous Power](#)
- [Gummel-Poon BJT](#)

You can find most of these examples in the software under the Examples directory in this location:

Tutorial/SDD_Examples_wrk/networks

Nonlinear Resistor

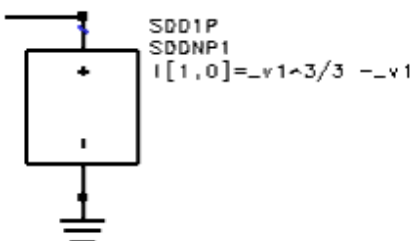
This section describes how to use SDDs nonlinear resistors with a cubic nonlinearity example. This example is under the Examples directory in the following location:

Tutorials/SDD_Examples_wrk/networks/Cubic

The nonlinear two-terminal resistor with constitutive relationship

$$i(v) = v^3/3 - v$$

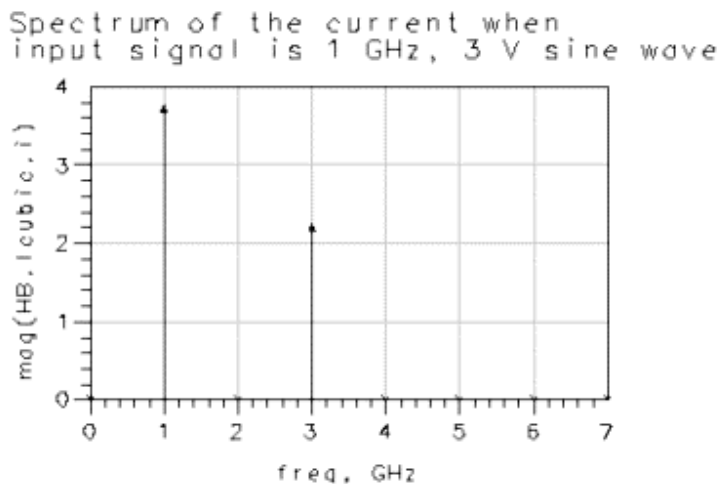
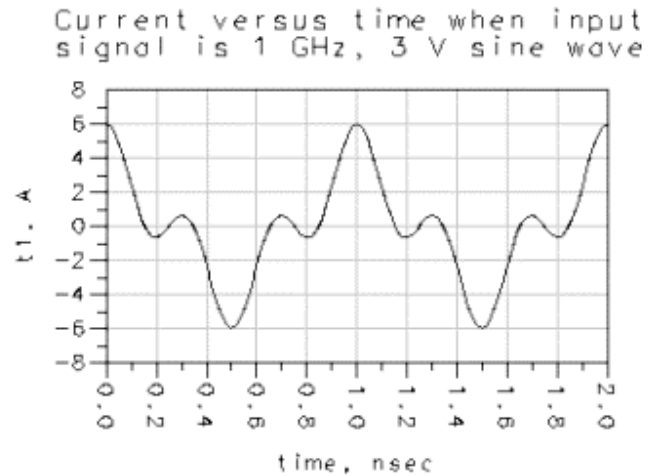
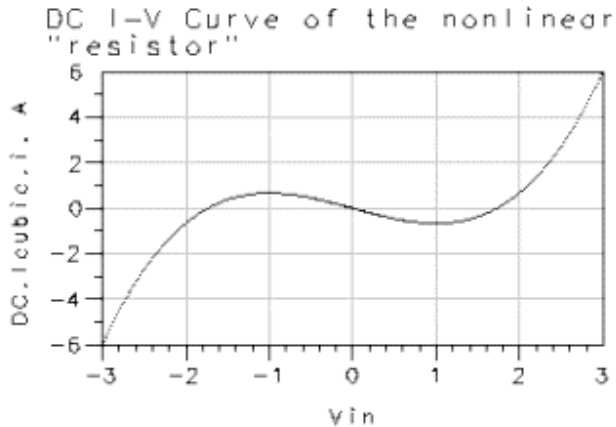
exhibits a negative resistance for small v , and is widely used in the study of oscillation theory. This two-terminal device can be modeled using a one-port SDD, shown below. Since this is a voltage-controlled resistor, the SDD is defined using an explicit equation.



With this setup, note the following points:

- This constitutive relationship specifies the current of port 1, and it is written as a function of the voltage at port 1.
- The Weight field is set to 0 to indicate that the weighting function is identically 1.

Results of DC and harmonic balance simulations on this component are shown in the following figure.



$$\text{Eqn } t1 = ts(\text{HB.icubic.i})$$

Simulation Results For the Nonlinear Cubic Resistor

The data displays show:

- A DC plot of current versus voltage showing the cubic nature of the resistor.
- The spectrum of the resistor current when a 1MHz, 3 V sinusoidal waveform is applied across the resistor. Note that the fundamental and the third harmonic are the only non-zero terms.
- Current versus time with the same waveform applied at the input.

Ideal Amplifier Block

This example is under the Examples directory in the following location:

Tutorials/SDD_Examples_wrk/networks/NonlinearAmp

A simple large-signal model for the gain of an ideal amplifier block can be expressed as

$$v_o = V_s \tanh(Av_i/V_s)$$

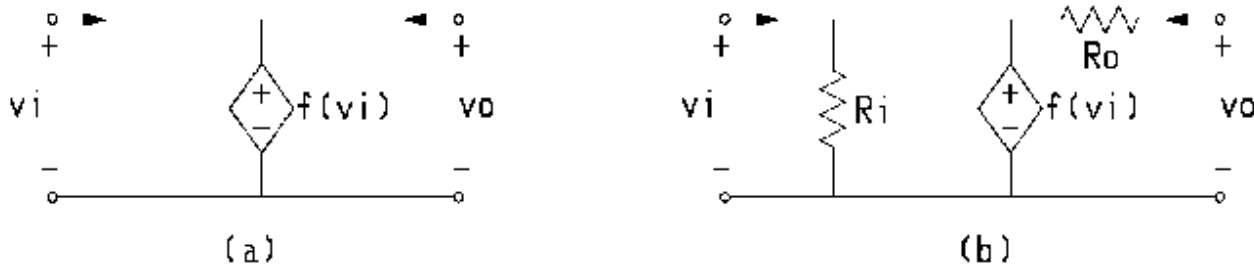
where:

- v_i is the input voltage
- v_o is the output voltage
- V_s is the power supply voltage
- A is the gain in the linear region

This relationship has the characteristics that the gain is A for small v_i , and that v_o saturates at $\pm V_s$, as shown in the following figure (a).

The amplifier is a two-port device, so one more equation is required to specify the constitutive relationship. In the case of the following figure (a), where the ideal amplifier has infinite input resistance and zero output resistance, you could use the above equation and the equation $i_i = 0$ to define the constitutive relationship.

To model the amplifier as shown in the following figure (b), with finite input resistance R_i and non-zero output resistance R_o , the equations will be different. The SDD in this example is based on this model.



Equivalent Circuit Model for an Ideal Saturating Amplifier

Current through the input resistance R_i can be expressed as:

$$i_i = (v_i/R_i).$$

You could use this equation directly as the equation for port 1, but then it would be impossible to set $R_i = \infty$. So, rewrite the explicit equation for port 1 using input

conductance G_i instead:

$$i_1 = G_i \cdot v_i$$

For port 2, the non-zero output resistance R_o is included in the model by adding a term to

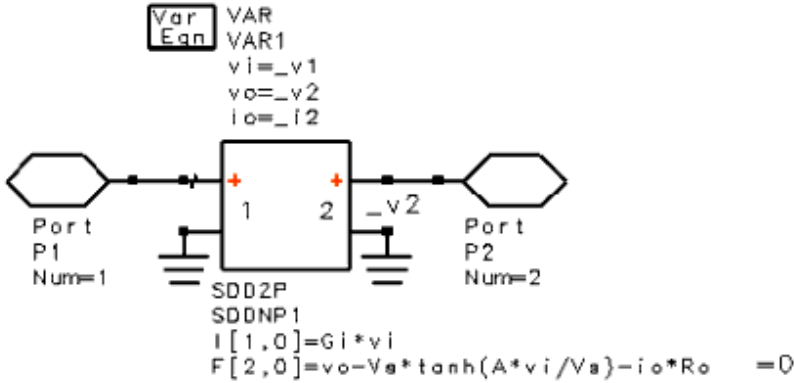
the equation $f(v_i) = V_s \tanh(Av_i/V_s)$ to account for the voltage drop across the output resistance:

$$v_o = V_s \tanh(Av_i/V_s) + i_o R_o$$

Note

We can use the port 2 current in this equation because the equation for port 2 is an implicit equation. Recall that when the equation for port n is implicit, the simulator appends the current through port n to the list of unknowns and, therefore, the value of $_{in}$ is available.

This model of an ideal amplifier as two-port SDD with the mixture of explicit and implicit equations is shown below.

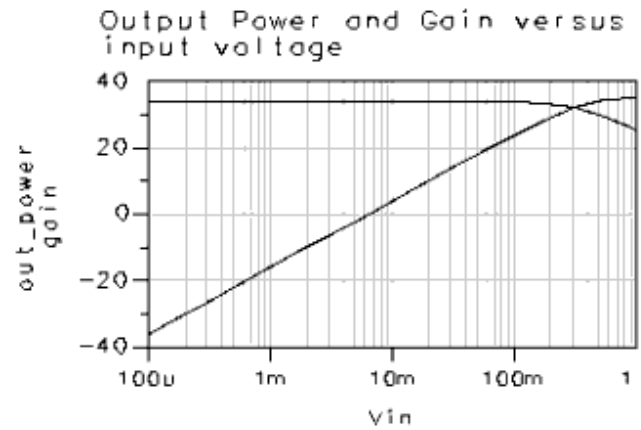
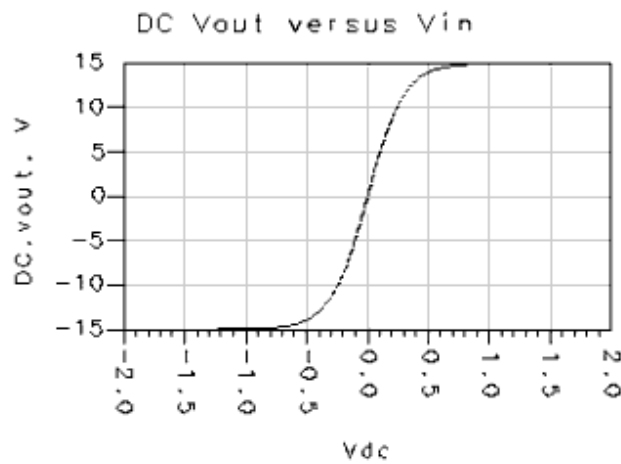


Note the following points:

- There are several parameters whose values are set by the user then passed to the device: G_i (input conductance), A (gain), V_s (saturated output voltage), and R_o (output resistance).
- $_{v1}$, $_{v2}$, and $_{i2}$ are assigned to variables (vi , vo , and io , respectively), and the variables are used in the SDD equations.
- The final form of the implicit equation for port 2 is written so that it equates to zero.

The SDD is simulated in the cell *TestAmp*. DC and harmonic balance simulation results are shown in the following figure.

- The first plot is a DC plot of v_o versus v_i .
- The second plot is harmonic balance results showing output power and gain as the amplifier saturates.



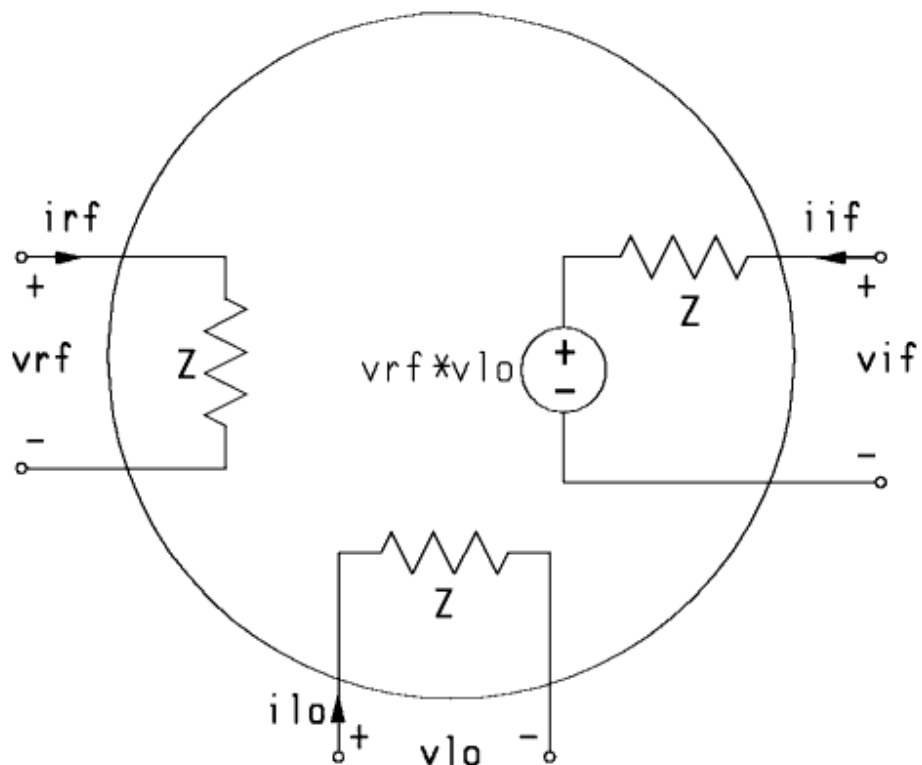
Simulation Results for the Ideal Saturating Amplifier

Ideal Mixer

This example is under the Examples directory in the following location:

Tutorials/SDD_Examples_wrk/networks/IdealMixer

The equivalent circuit for an ideal mixer is shown in the following figure.



Equivalent Circuit for an Ideal Mixer

Equivalent Circuit for an Ideal Mixer

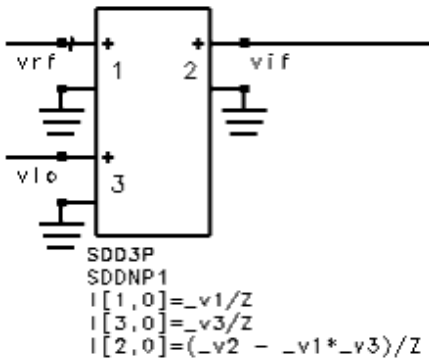
The ideal mixer is a three-port device, so three equations are required to define its constitutive relationship. Based on the circuit above, the following three equations can be used to represent the current at each port:

$$i_{rf} = v_{rf} / Z$$

$$i_{lo} = v_{lo} / Z$$

$$i_{if} = (v_{if} - v_{rf} v_{lo}) / Z$$

These equations are voltage-controlled and can be implemented using explicit SDD equations. The SDD is shown next.

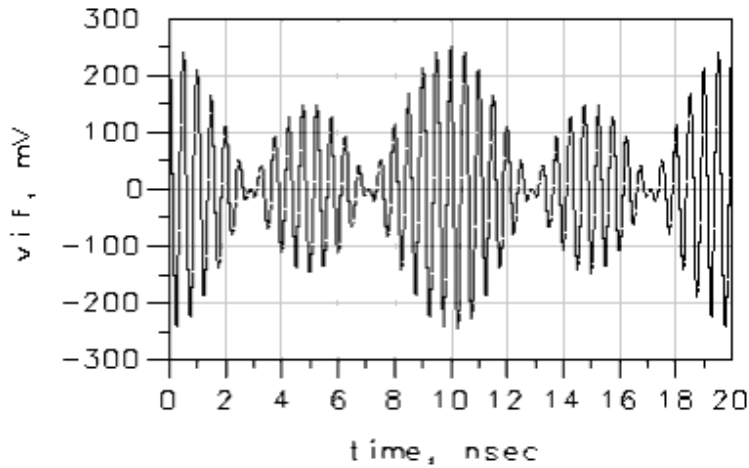


In this setup `_v1`, `_v2`, and `_v3` were used in the equations. Each port has a named node, so the voltages will appear in the data display.

RF-LO-IF Example

An RF input of 1V at 3 GHz and an LO input of 1V at 4 GHz yields an IF output of 0.25V at 1 GHz and 7 GHz, provided the IF output is matched (terminated in Z). The one scaling down by a factor of two comes from the ideal mixing process, while the other comes from the voltage being split over the two Z s.

The following figure shows the results of a transient analysis simulation of the mixer. It shows the amplitude modulation effects in the time waveform of v_{if} . For this simulation, v_{rf} is a sinusoid at 100 MHz with a DC offset, and v_{lo} is a sinusoid at 2 GHz.



Simulation Results for the Ideal Mixer

Nonlinear Capacitors

So far, all of the examples have dealt with nonlinear resistors. This section describes nonlinear capacitors.

A nonlinear, voltage-controlled capacitor is defined in terms of its charge-voltage, or $q - v$, relationship

$$q = Q(v)$$

For example, the $q - v$ relationship for a linear two-terminal capacitor is

$$q = Cv$$

which, when differentiated with respect to time, yields the more familiar capacitor equation

$$i = C(v) \frac{dv}{dt}$$

To use the SDD to model a nonlinear voltage-controlled capacitor, note that given a nonlinear charge $Q(v)$, the current is

$$i = \frac{d}{dt} Q(v)$$

This is a voltage-controlled expression for the current. It differs from the constitutive relationship of a voltage-controlled resistor because it contains a time derivative.

The time derivative is implemented in the SDD by specifying weighting function number 1. Weighting function number 1 is predefined as $j\omega$ which is the frequency-domain version

of the time derivative.

Obtaining Charge From Capacitance

Often the equation for a nonlinear capacitor is specified not in terms of charge, but in terms of a nonlinear capacitance $C(v)$ where

$$i = C(v) \frac{d}{dt}$$

Given this representation, the charge function is obtained by integrating the capacitance

$$Q(v) = \int_v C(\hat{v}) d\hat{v} + Q_o$$

where we have explicitly included the arbitrary constant of integration Q_o .

If for some reason, the charge cannot be calculated, then the alternative technique presented in [Alternative Implementation of a Capacitor](#) can be used to implement the capacitor.

Multi-port Capacitors

A nonlinear voltage-controlled two-port capacitor is usually defined by a capacitance matrix

$$C(v_1, v_2) = \begin{bmatrix} C_{11}(v_1, v_2) & C_{12}(v_1, v_2) \\ C_{21}(v_1, v_2) & C_{22}(v_1, v_2) \end{bmatrix}$$

The capacitor currents are given by

$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = C(v_1, v_2) \frac{d}{dt} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

The charge for a two-port capacitance is defined as the function $Q(v_1, v_2)$ such that $C(v_1, v_2)$ is the derivative (that is, Jacobian) of $Q(v_1, v_2)$. It follows that $Q(v_1, v_2)$ exists if and only if

$$\frac{\partial C_{11}}{\partial v_2} = \frac{\partial C_{12}}{\partial v_1}$$

and

$$\frac{\partial C_{21}}{\partial v_2} = \frac{\partial C_{22}}{\partial v_1}$$

If $Q(v_1, v_2)$ does not exist, then the technique presented in [Alternative Implementation of a Capacitor](#) can be used to implement the capacitor.

Full Model Diode, with Capacitance and Resistance

This example is under the Examples directory in the following location:

Tutorials/SDD_Examples_wrk/networks/SDD_Diode

Capacitance. The junction capacitance of a reverse-biased pn diode may be written as

$$C_r(v) = C_o \sqrt{\frac{V_o}{V_o - v}} \quad v < V_o$$

The subscript r signifies reverse bias.

To develop this expression into an equation that can be used in an SDD, you integrate $C_r(v)$ with respect to v to get an expression for the charge:

$$Q_r(v) = -2C_o \sqrt{V_o(V_o - v)} \quad v < V_o$$

where the arbitrary constant of integration is chosen so that $Q_r(V_o) = 0$.

There is a limitation to this equation because it is valid only for $v < V_o$. Though it is useful in applications where the diode is always reverse biased (for example, a varactor diode), it is not suitable for a general harmonic-balance analysis (or a DC analysis, for that matter) where the bias voltage may exceed V_o .

A better diode model has the charge model extended into the forward-biased region, plus resistance. Capacitance is described next, followed by resistance and the SDD implementation. Besides yielding a valuable result, this example also highlights some useful techniques for ensuring the continuity of charge and its derivative.

To increase the range of operation of the model, you can extend the capacitance into the region $v > V_o$ using a linear extrapolation. To do this, choose α such that $0 < \alpha < 1$.

Let the previous $C_r(v)$ equation be valid for $v < \alpha V_o$, and for $v > \alpha V_o$ use

$$C_f(v) = C_r(\alpha V_o) + C'_r(\alpha V_o)(v - \alpha V_o)$$

where:

- $C'_r(v)$ is the derivative of $C_r(v)$ with respect to v
- The subscript f signifies forward bias
- C_f is a linear extension of C_r that matches the value and slope of C_r at $v = \alpha V_o$

This definition of C_f ensures that, when joined with C_r , the capacitance and its derivative are continuous. The boundary between reverse and forward bias is chosen to be αV_o instead of V_o because the slope of C_r at V_o is infinite.

The next step is to integrate $C_f(v)$ to obtain

$$Q_f(v) = (v - \alpha V_o)(C_r(\alpha V_o) + C'_r(\alpha V_o)(v - \alpha V_o)/2) + Q_r(\alpha V_o) \quad v \geq \alpha V_o.$$

where the constant of integration is chosen so that $Q_f(\alpha V_o) = Q_r(\alpha V_o)$. This equation can be rewritten as

$$Q_f(v) = \frac{C_o}{\sqrt{1-\alpha}} \left(v - \alpha V_o + \frac{(v - \alpha V_o)^2}{4V_o(1-\alpha)} \right) + Q_r(\alpha V_o) \quad v \geq \alpha V_o.$$

The overall expression for the junction charge is given as

$$Q(v) = \begin{cases} Q_r v & \text{if } v < \alpha V_o \\ Q_f v & \text{if } v \geq \alpha V_o \end{cases}$$



Note

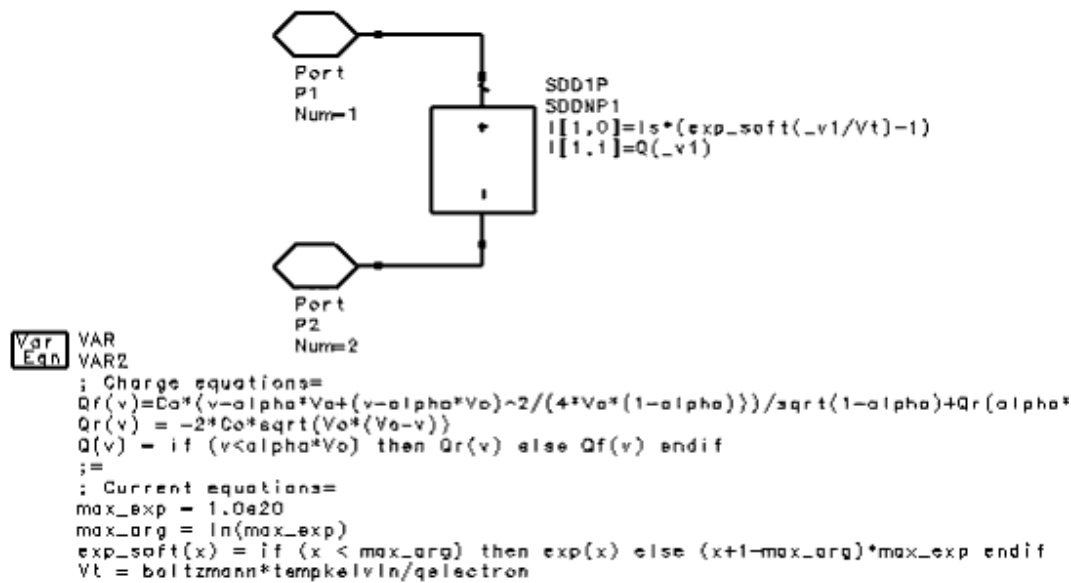
$Q(v)$ and its derivatives are guaranteed continuous due to the definition of $C_f(v)$ and due to the choice of the constant of integration for $Q_f(v)$.

Resistance. The equation for the resistive behavior of a pn junction is the ideal diode equation

$$i = I_s (\exp(v/V_T) - 1).$$

Thus, total diode current has two components, one from the ideal diode equation and one from the charge. This is handled in the SDD by specifying two equations for the current of port one, one using weighting function number 0 and the other using weighting function number 1.

Implementation. The SDD implementation is shown next.

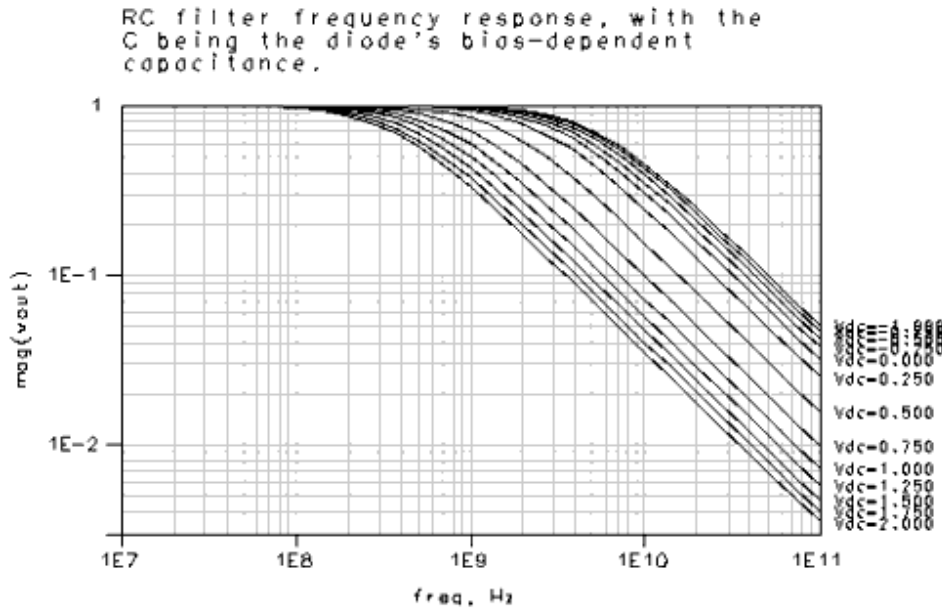


Note the following points:

- The current in the diode is based on two SDD equations:
 - The first equation models the resistive behavior of the diode. It uses expressions listed in the Var Egn component under Current equations. These include the variables *max_exp*, *max_arg*, the function *exp_soft(x)*, and the variable *Vt*. They determine what value *Is* is multiplied by. *exp_soft* is the soft exponential function and is used to prevent overflow problems when taking the exponent of a large number. It is the same as a normal exponential except it becomes a linear extrapolation when its argument is such that the normal exponential would exceed *max_exp*.
 - The second equation models the charge. It uses the expressions listed in the Var Egn component under Charge equations. The value of *_v1* is passed to the function *Q(v)*, where it is evaluated and the result is returned to the SDD. There are several parameters with user-defined values, which also enter into the calculations: *Is* (), *Co* (), *Vo* (), and *alpha* () (these value are passed from *TestDiode*).
- A weighting function is used in the second SDD equation. It is important to understand how the weighting function is used by the SDD and is reviewed here.
 - The spectrum for the port voltage *_v1* is inverse Fourier transformed into the time domain.
 - The constitutive relation (in this case, $-2*CO*sqrt(VO*(VO-vv))$) is evaluated point-by-point in the time domain.
 - The resulting waveform (which is the charge for port one) is Fourier transformed into the frequency domain.
 - The weighting function (in this case, *jw*) is applied in the frequency domain. The result is the spectrum of the port current *_i1*.
- When two explicit equations are specified for a single port, the SDD calculates a spectrum representing the (weighted) result of the first equation, calculates a spectrum representing the (weighted) result of the second equation, and then sums the two spectra to get the final spectrum for the port current.

The SDD is simulated in the cell *TestDiode*. This design uses the diode capacitance as the

C in an RC circuit. It also allows the independent adjustment of the diode bias voltage. The following figure shows the frequency response of the RC circuit as the bias voltage is varied from -1 to 2 V.



Full Varactor Diode Model Results with $C_0 = 1 \text{ pF}$, $V_0 = 0.65\text{V}$, and $\alpha = 0.7$

Nonlinear Inductors

A nonlinear current-controlled inductor is defined in terms of its flux-current, or ϕ - i , relationship

$$\phi = \Phi(i).$$

For example, the ϕ - i relationship for a linear two-terminal inductor is

$$\phi = Li$$

which, when differentiated with respect to time, yields the more familiar inductor equation

$$v = L \frac{di}{dt}.$$

To model a current-controlled nonlinear inductor, differentiate

$$\phi = \Phi(i)$$

with respect to time to obtain

$$v = \frac{d}{dt}\Phi(i)$$

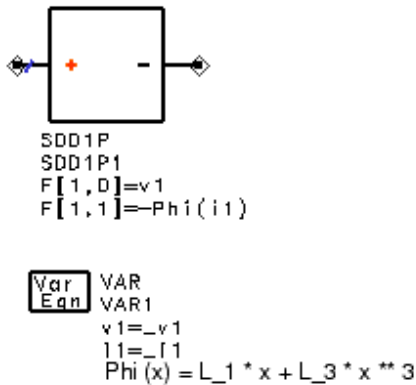
which can be rewritten as

$$v - \frac{d}{dt}\Phi(i) = 0.$$

This expression can be implemented using an implicit representation. For example, the SDD implementation for the nonlinear inductor specified by

$$\Phi(i) = L_1 i + L_3 i^3$$

is



Note

This is a good example of a case when using weighting functions with the implicit representation makes sense.

Note that Advanced Design System also includes a built-in nonlinear inductor (*NonlinL*) available from the *Eqn Based-Nonlinear* component palette.

Obtaining Flux From Inductance

Often the equation for a nonlinear inductor is specified not in terms of flux, but in terms of a nonlinear inductance $L(i)$ where

$$v = L(i) \frac{di}{dt}.$$

Given this representation, the flux function is obtained by integrating the inductance

$$\Phi(i) = \int_i L(\hat{i}) d\hat{i} + \Phi_0$$

where we have explicitly included the arbitrary constant of integration ϕ_0 .

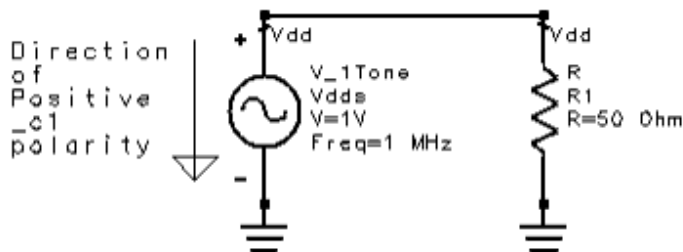
Controlling Current, Instantaneous Power

This example is under the Examples directory in the following location:

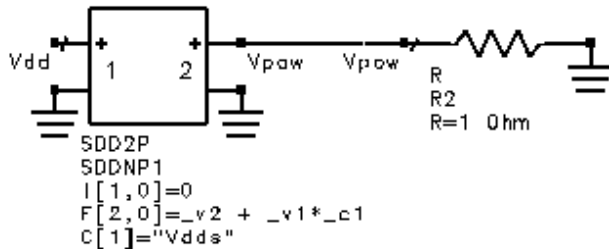
Tutorials/SDD_Examples_wrk/networks/RemCC

This example illustrates how to use a current as part of an SDD equation, where the current is from another device in the circuit. For more background on controlling currents and how to implement them, refer to [Controlling Currents](#) and [Defining a Controlling Current](#).

In this example, an SDD is used to calculate the instantaneous power dissipated through resistor R1. The circuit containing R1 is shown here.



Making the power calculation requires both the voltage across R1 and the current through R1. These values are supplied to the SDD in the following manner:



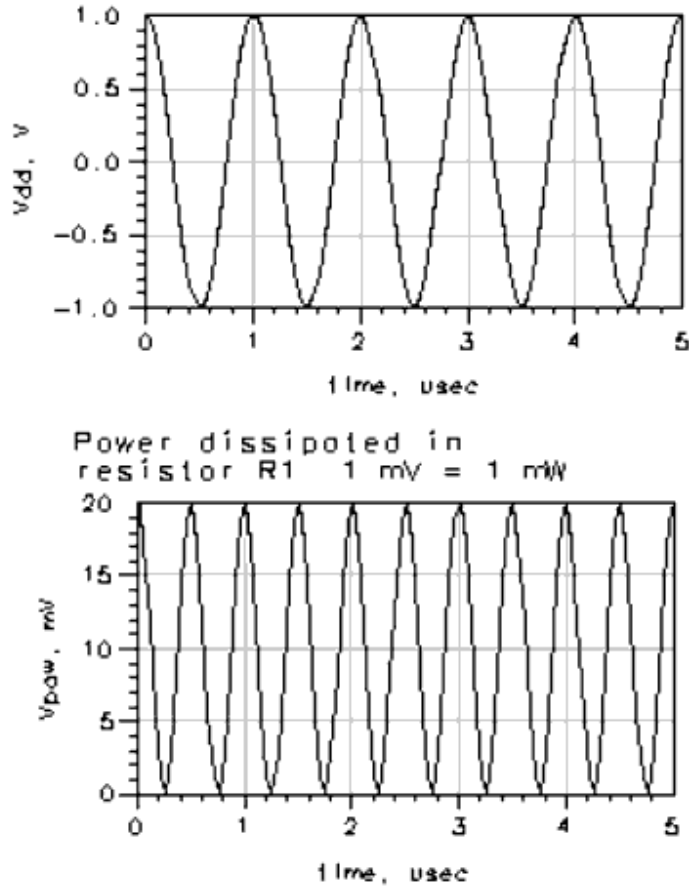
- The voltage across R1, labeled Vdd , is applied to port 1 of the SDD. Note that the current at this port is set to zero.
- The current through R1 is specified by using the current through the voltage source $Vdds$, and reversing polarity. Recall that only the current through either a voltage source or current probe can be used as a controlling current. The instance name of the component is used to specify the controlling current, as shown in the SDD illustration. In a more complex circuit, you might consider adding a current probe.
- Although the equation to find power dissipated in R1 is simply $Vdd * _c1$, it must be written in a form that is suitable for the SDD. The first step is to substitute $_v1$ for Vdd . Then note that if:

$$_v2 = -_v1 * _c1$$

and by using an implicit equation, the equation

$$_v2 + _v1*_c1$$

can be used to define port 2 of the SDD. Then use a named node (*Vpow*) to save the power to the dataset. The graphs of *Vdd* and the instantaneous power *Vpow* are shown below.

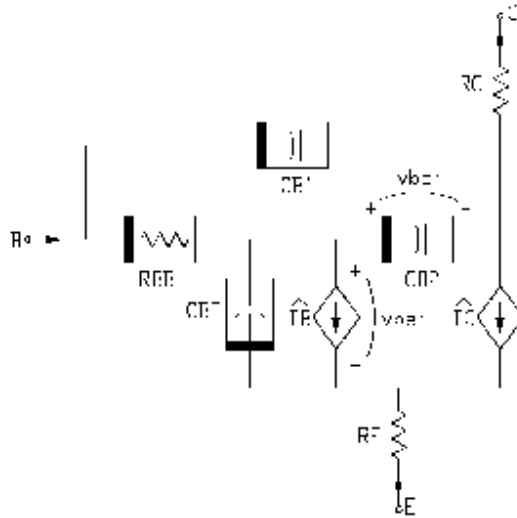


Gummel-Poon BJT

This example is under the Examples directory in the following location:

Tutorials/SDD_Examples_wrk/networks/GumPoon

The following figure shows the equivalent circuit model for the Gummel-Poon bipolar junction transistor (BJT). The associated current and capacitance equations follow.



Equivalent Circuit Model for the Gummel-Poon BJT



Note

The Gummel-Poon model shown here is only an illustrative example of SDD. For a Gummel-Poon BJT model that is fully tested and qualified, please use the Devices-BJT component palette in Advanced Design System.

Current Equations

$$\hat{i}_c = \frac{J_s}{Q_b} \left(e^{\left(\frac{v_{bci}}{N_f V_t} \right)} - e^{\left(\frac{v_{bci}}{N_r V_t} \right)} \right) - \frac{J_s}{B_r} \left(e^{\left(\frac{v_{bci}}{N_r V_t} \right)} - 1 \right) - J_{lc} \left(e^{\left(\frac{v_{bci}}{N_c V_t} \right)} - 1 \right)$$

$$\hat{i}_c = \frac{J_s}{B_f} \left(e^{\left(\frac{v_{bci}}{N_f V_t} \right)} - 1 \right) - \frac{J_s}{B_r} \left(e^{\left(\frac{v_{bci}}{N_r V_t} \right)} - 1 \right) - J_{le} \left(e^{\left(\frac{v_{bci}}{N_c V_t} \right)} - 1 \right) + J_{lc} \left(e^{\left(\frac{v_{bci}}{N_c V_t} \right)} - 1 \right)$$

$$R_{bb} = 3 \cdot (rb - rbm) \cdot (\tan(Z) - Z) / (Z \cdot \tan(Z)^2) + rbm$$

where

$$Q_b = \frac{Q_1}{2}(1 + \sqrt{1 + 4Q_2})$$

$$Q_1 = \left(1 - \frac{V_{bci}}{V_{bf}} - \frac{V_{bei}}{V_{br}}\right)^{-1}$$

$$Q = \frac{J_s}{J_{bf}} \left(e^{\left(\frac{V_{bei}}{N_f V_t}\right)} - 1 \right) + \frac{J_s}{B_r} \left(e^{\left(\frac{V_{bci}}{N_r V_T}\right)} - 1 \right)$$

Capacitance Equations

$$C_{b1} = (1 - X_{cjc}) C_{jc} \left(1 - \frac{V_b - V_{ci}}{V_{jc}}\right)^{-M_{jc}}$$

$$C = \frac{T_r J_s}{N_r V_t} e^{\left(\frac{V_{bci}}{V_T}\right)} + X_{cjc} C_{jc} \left(1 - \frac{V_{bci}}{V_{jc}}\right)^{-M_{jc}}$$

$$C_{be} = \frac{\partial}{\partial v_{bei}} \left(\frac{T_{ff} J_s}{Q_b} \left(e^{\left(\frac{V_{bei}}{N_f V_t}\right)} - 1 \right) \right) + C_{je} \left(1 - \frac{V_{bei}}{V_{je}}\right)^{-M_{je}}$$

where

$$T_{ff} = -TF \left(1 + X_{tf} e^{\left(\frac{v_{bci}}{1.44 V_{tf}}\right)} \left(\frac{I_f}{I_f + J_{tf}} \right)^2 \right)$$

$$I_f = J_s \left(e^{\left(\frac{V_{bci}}{N_f V_t}\right)} - 1 \right)$$

Note : Junction capacitances of the form

$$C \left(1 - \frac{v}{V_o}\right)^{-M}$$

change to the form

$$\frac{C}{(1-F_c)^M} \left(1 + \frac{M}{V_o(1-F_c)} (v - F_c V_o) \right)$$

when $v > F_c V_o$. Here, $0 < F_c < 1$.

Adding the Nonlinear Base Resistance

In the full Gummel-Poon model, the base resistance R_{bb} is a nonlinear resistance that depends on i_b . When the base resistance is nonlinear, it cannot be modeled by a discrete resistor-it must be included in the SDD equations.

The constitutive relationships are:

$$i_b = (v_b - v_{bi})/R_{bb}$$

$$0 = (v_{bi} - v_b)/R_{bb} + i_b(v_{bei}, v_{bci}) + \frac{d}{dt}(Q_{be} + Q_{b2})$$

$$\hat{i}_c = \hat{i}_c(v_{bei}, v_{bci}) - \frac{d}{dt} Q_{b2}$$

$$i_e = -\hat{i}_b(v_{bei}, v_{bci}) - \hat{i}_c(v_{bei}, v_{bci}) - \frac{d}{dt} Q_{be}.$$

Adding the Split Base-Collector Charge

Now that the nonlinear base resistance has been modeled, adding the split base-collector capacitance is straight-forward. First, modify the equation for Q_{b2} to account for X_{cjc} . Second, insert the equation for Q_{b1} . Finally, add the time derivative of Q_{b1} to i_b and subtract it from i_c :

$$i_b = (v_b - v_{bi})/R_{bb} + \frac{d}{dt} Q_{b1}$$

$$0 = (v_{bi} - v_b)/R_{bb} + i_b(v_{bei}, v_{bci}) + \frac{d}{dt}(Q_{be} + Q_{b2})$$

$$i_c = \hat{i}_c(v_{bei}, v_{bci}) - \frac{d}{dt}(Q_{b1} + Q_{b2})$$

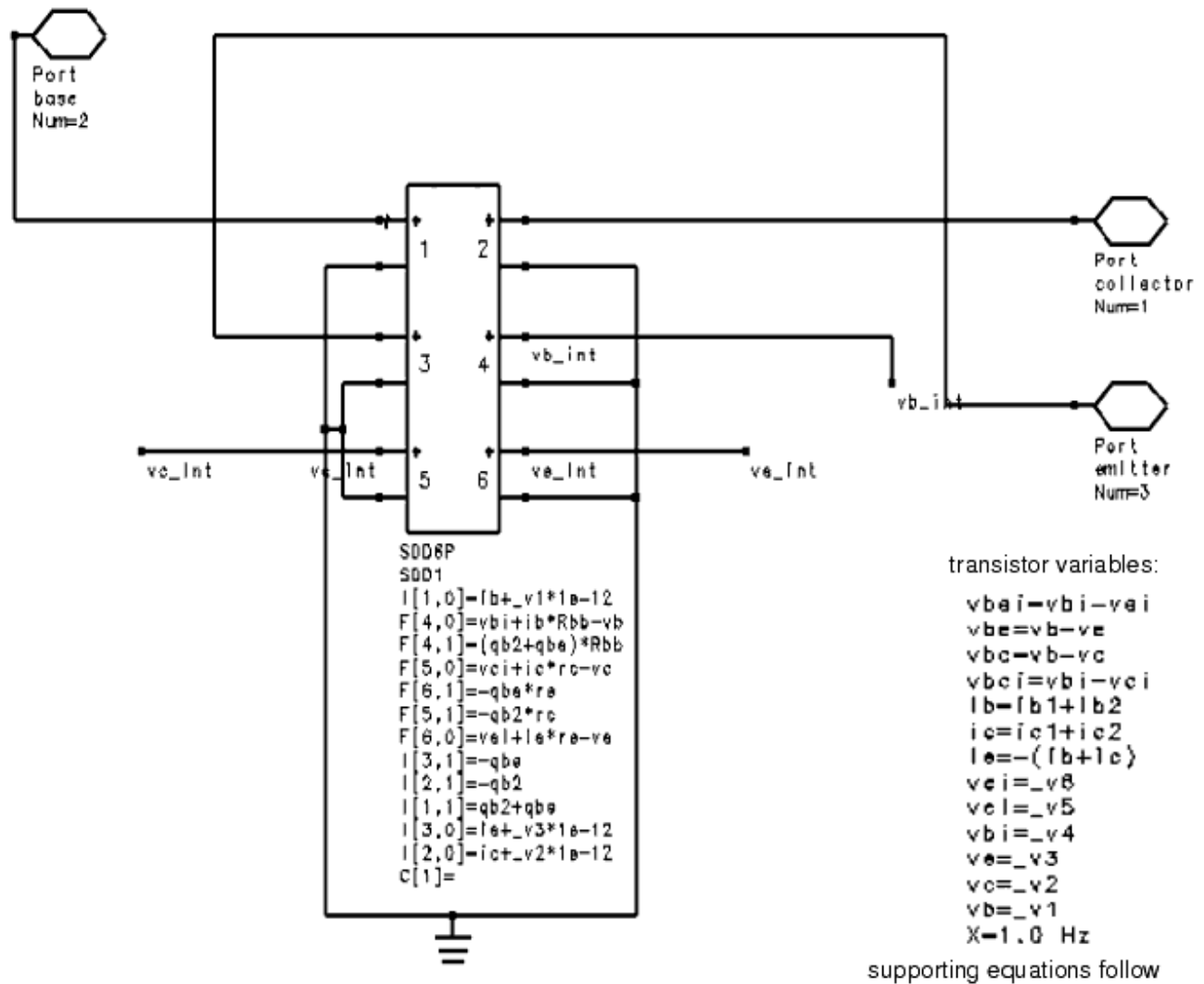
$$i_e = -\hat{i}_b(v_{bei}, v_{bci}) - \hat{i}_c(v_{bei}, v_{bci}) - \frac{d}{dt} Q_{be}.$$

SDD Implementation

This implemented SDD is under the Examples directory in the following location:

Tutorials/SDD_Examples_wrk/GumPoon

For optimal viewing, you should open the cell. The components and equations are shown below.




```

Var
  _VAR1
  ;temperature variables and equations=
  vje_T=vje
  Br_T=br
  Eg(t)=eg0-7.02e-4*(t+TzeroC)^2/(1106+(t+TzeroC))
  Bf_T=bf
  eps=1e-9
  Js_T=Js*TempRatio~x11 * exp_soft(Eg(Tamb)*((Tj-Tamb)/((Tamb+TzeroC)*vt)))
  TempRatio=(Tj+TzeroC)/(Tamb+TzeroC)
  TzeroC=273.15
  Tj=Tamb
  ;=
  ;non-linear base resistance=
  lbb=if (lb < 1pA) then 1pA else lb endif
  Z=(1 + sqrt(144*ibb/(pi^2*jrb) + 1))/((24/pi^2)*sqrt(ibb/jrb))
  Rbb=3*(rb - rbm)*(tan(Z) - Z)/(Z*tan(Z)^2 + rbm)
  ;=
  ;base-collector charge=
  qbc=tr*diode(vbci, Js_T, nr) - cjc*vjc_T*((1-vbci/vjc_T)^(1-mjc))/(1-mjc)
  qb2=qbc
  ;=
  ;base-emitter charge=
  lff=diode(vbei, Js_T, nf)
  TFF=1*(1+xtf*(exp(vbci/(1.44*vtf))*((lff/(lff+jtf))^2)))
  qbe=diode(vbei, TFF*Js/QB, nf) - cje*vje_T*((1-vbei/vje_T)^(1-mje))/(1-mje)
  ;=
  ;base current equations=
  ib2=diode(vbei, jle, na) + diode(vbci, jlc, nc)
  lb1=diode(vbei, Js_T/Bf_T, nf) + diode(vbci, Js_T/Br_T, nr)
  ;=
  ;collector current equations=
  ic2=-diode(vbci, Js_T/Br_T, nr) - diode(vbci, jlc, nc)
  ic1=diode(vbci, Js_T/QB, nf) - diode(vbci, js/QB, nr)
  ;=
  ;intermediate variables for collector current=
  Q1=1/(1 - vbci/vbf - vbei/vbr)
  Q2=diode(vbci, Js_T/jbf, nf) + diode(vbci, Js_T/jbr, nr)
  QB=(Q1/2)*(1 + sqrt(1+4*Q2))
  ;=
  ;useful general equations=
  vt=boltzmann*(Tamb+TzeroC)/qelectron
  diode(v,Is,n)=Is*(exp_soft(v/(n*vt)) - 1)
  exp_max=ln(1e18)
  exp_soft(x)=if (x < exp_max) then exp(x) else (x+1-exp_max)*exp(exp_max) end

```

Note the following points.

- Each port has two equations, one for the current and one for the charge.
- The capacitance equations were integrated to obtain charge equations:
 - The integration is simplified for the first term of Cbe since the first term is a partial derivative, the integration and partial derivative effectively cancel.
 - The integration is simplified for the first term in Cbc since the first term is an exponential, and integration of an exponential is another exponential.
 - The other charges are similar in form to the charge given earlier in the section, [Full Model Diode, with Capacitance and Resistance](#).
- The *diode()* and *charge()* functions are used to make the equations more readable and to eliminate the duplication of common expressions.
- Except for one difference, the SDD BJT presented here is *identical* to the compiled BJT model built-in to the simulator (in the simulator, the values of Vje and Vjc are adjusted to reflect the bandgap characteristics of silicon).
- The SDD BJT uses about 55 equations. The built-in BJT model requires over 4500 lines of C code.
- The SDD BJT was written in about one day, and debugged in about one day. The built-in BJT model required about two weeks to write and another two weeks to debug.

Examples Summary

- A *voltage-controlled nonlinear resistor* is described by its i-v relation
 $i = I(v)$
- A two-terminal voltage-controlled nonlinear resistor $i = I(v)$ is implemented by
 $I[1,0] = _v1$
- A *general nonlinear resistor* is described by an implicit i-v relation
 $f(i, v) = 0$.
- A general two-terminal nonlinear resistor $f(i, v) = 0$ is implemented by
 $I[1,0] = f(_i1, _v1)$
- A *voltage-controlled nonlinear capacitor* is described by its q-v relation
 $q = Q(v)$.
- A two-terminal voltage-controlled nonlinear capacitor $q = Q(v)$ is implemented by
 $I[1,1] = Q(_v1)$
- A two-terminal voltage-controlled device with resistance $i = I(v)$ and charge $q = Q(v)$ is implemented by
 $I[1,0] = I(_v1)$
 $I[1,1] = Q(_v1)$
- If a capacitor is specified by a nonlinear capacitance $C(v)$ where
 $i = C(v) \frac{dv}{dt}$,

then the corresponding charge is given by

$$Q(v) = \int_v C(\hat{v}) d\hat{v} + Q_o$$

where Q_o is the arbitrary constant of integration.

- A *current-controlled nonlinear inductor* is described by its ϕ -i relation
 $\phi = \Phi(i)$.
- * A two-terminal current-controlled nonlinear inductor $\phi = \Phi(i)$ is implemented by
 $I[1,0] = _v1$
 $I[1,1] = -\phi(_i1)$
- If an inductor is specified by a nonlinear inductance $L(i)$ where
 $v = L(i) \frac{di}{dt}$,

then the corresponding flux is given by

$$\Phi(i) = \int_i L(\hat{i}) d\hat{i} + \Phi_o$$

where Φ_o is the arbitrary constant of integration.

- SDD models are easier to write and debug than compiled models, but they are less efficient during a simulation.

Modified Nodal Analysis

Advanced Design System uses nodal analysis to form the circuit equations. Nodal analysis is based on Kirchhoff's current law (KCL) which states that for each node, the sum of the currents incident to the node is zero.

Suppose a circuit has $n+1$ nodes and b branches. Let i be the vector of branch currents.

Then KCL can be expressed by the equation

$$A\mathbf{i} = \mathbf{0}$$

where A is an $n \times b$ matrix called the *node incidence matrix*. The entries in A are given by

$$a_{ij} = \begin{cases} 1 & \text{if branch } j \text{ enters node } i \\ -1 & \text{if branch } j \text{ leaves node } i \\ 0 & \text{otherwise} \end{cases}$$

In nodal analysis, KCL is not applied to the ground node (such an equation yields no independent information) which explains why A has only n rows. If all the devices in the circuit are voltage controlled, that is, if the port currents of each device are completely determined by the port voltages of that device, then the branch current vector \mathbf{i} can be written as

$$\mathbf{i} = \mathbf{g}(\mathbf{v})$$

where \mathbf{v} represents the vector of n node voltages and \mathbf{g} is a map from \mathbb{R}^n to \mathbb{R}^b . Substituting this equation into the KCL equation yields the node analysis equation

$$\mathbf{G}(\mathbf{v}) = \mathbf{0}$$

where \mathbf{G} is a map from \mathbb{R}^n to \mathbb{R}^n defined by $\mathbf{G}(\mathbf{v}) = \mathbf{A}\mathbf{g}(\mathbf{v})$.

When a circuit contains devices that are not voltage controlled (a voltage source or an inductor, for example), it is impossible to write KCL in terms of the node voltages alone—some additional variables must be used. In *modified nodal analysis*, the branch currents of the non-voltage-controlled devices are retained as variables. Thus KCL can be written as

$$\hat{\mathbf{G}}(\mathbf{v}, \mathbf{i}_b) = \mathbf{0}$$

where \mathbf{i}_b is the vector of the n_b branch currents of the non-voltage-controlled devices and $\hat{\mathbf{G}}$ is a map from \mathbb{R}^{n+n_b} to \mathbb{R}^n .

Since there are now $n+n_b$ unknowns, n_b additional equations must be appended to the node equations. These additional equations are the constitutive relationships of the n_b non-voltage-controlled branches

$$\hat{\mathbf{f}}(\mathbf{v}, \mathbf{i}_b) = \mathbf{0}$$

The resulting augmented nodal equations are the modified nodal analysis equations

$$\mathbf{F}(\mathbf{v}, \mathbf{i}_b) = \begin{pmatrix} \hat{\mathbf{G}}(\mathbf{v}, \mathbf{i}_b) \\ \hat{\mathbf{f}}(\mathbf{v}, \mathbf{i}_b) \end{pmatrix} = \mathbf{0}$$

Alternative Implementation of a Capacitor

Suppose you have a nonlinear capacitance that cannot be integrated to get the corresponding charge function. One example is a capacitance that is table-driven from experimentally obtained data. Another case is a two-port capacitor

$$C(v_1, v_2) = \begin{bmatrix} C_{11}(v_1, v_2) & C_{12}(v_1, v_2) \\ C_{21}(v_1, v_2) & C_{22}(v_1, v_2) \end{bmatrix}$$

where there does not exist a charge $Q(v_1, v_2)$ such that $C(v_1, v_2)$ is the Jacobian of $Q(v_1, v_2)$. In these cases, the capacitor can still be implemented using an SDD. Consider the one-port nonlinear capacitance $C(v)$. By definition,

$$i = C(v) \frac{dv}{dt}$$

There is no way to implement this equation directly using an SDD because it involves the product of a derivative. To bypass this problem, create an intermediate variable $dv_dt = dv/dt$. Then the capacitor is described by the equations

$$i = C(v) dv_dt$$

$$dv_dt = \frac{dv}{dt}.$$

There is one problem with implementing these equations directly. In the frequency domain, the time derivative of v is

$$dv_dt = j2\pi f v.$$

Considering harmonic frequencies, f can be as high as 500 GHz. With such a large value of f , a $1\mu V$ change in v produces a 3 MV change in dv_dt . This high sensitivity can cause convergence difficulties for the system. To eliminate the problem, scale by a nominal frequency value of 1 GHz.

$$f_{nom} = 1\text{GHz}$$

$$i = C(v) f_{nom} dv_dt$$

$$dv_dt = \frac{1}{f_{nom}} \frac{dv}{dt}.$$

Note that even though i is proportional to $f_{nom} dv_dt$, i is not overly sensitive to dv_dt because f_{nom} is multiplied by $C(v)$ which is typically on the order of $1/f_{nom}$.

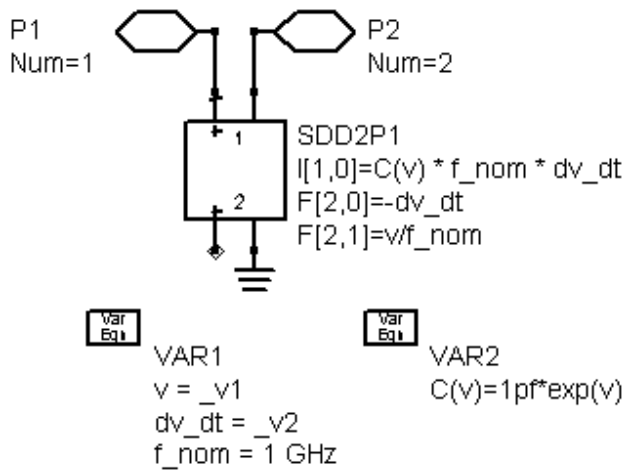
The scaled formulation of the capacitance is implemented by the SDD using the following equations:

$$\begin{aligned} I[1,0] &= C(v) * f_{nom} * dv_dt \\ F[2,0] &= -dv_dt \\ F[2,1] &= v / f_{nom} \end{aligned}$$

and these VAR equations:

$$\begin{aligned} v &= _v1 \\ dv_dt &= _v2 \\ f_{nom} &= 1\text{ GHz} \\ C(v) &= 1\text{pf} * \exp(v) \end{aligned}$$

This SDD can be found in *Examples/Tutorials/SDD_Examples_wrk* as *SDD_cap*. An alternate implementation, *SDD_cap2*, can also be found in the workspace.



Error Messages

If an SDD has not been implemented correctly, it will generate errors. The errors will be similar to the ones listed here.

Expression error: [error __ message].

An error has occurred while parsing or differentiating an expression.

$h[0]$ and $h[1]$ are predefined.

Weighting functions 0 and 1 have been redefined. This is not allowed.

Illegal state variable $_in$.

$_in$ has been used, but there are not n ports.

Illegal state variable $_vn$.

$_vn$ has been used, but there are not n ports.

Improper frequency dependence in sdd $_f$ parameters.

One or more of the implicit relationships depends on *freq* or *omega*. Frequency dependence is not allowed.

Improper frequency dependence in sdd $_i$ parameters.

One or more of the explicit relationships depends on *freq* or *omega*. Frequency dependence is not allowed.

Port equation cannot be both i and f type.

At least one of the ports has both an explicit and an implicit expression. If more

than one expression is used for a port, all expressions for the port must be of the same type, that is, all explicit or all implicit.

Port n is missing a corresponding equation.

No constitutive relationship has been specified for port n .

SYM error: [error message].

An error has occurred while evaluating an SDD expression or its derivative.
(SYM is the name of the system symbolic expression handler.)

Custom Modeling with Frequency-Domain Defined Devices

As CAE plays a larger role in the design cycle of RF and microwave circuits and subsystems, it is important for CAE design systems to satisfy the modeling needs of the engineer at both the device level and the subsystem level. As communication applications continue to increase, it is no longer possible to satisfy all modeling needs with standard, preconfigured models. Thus, Advanced Design System enables users to define their own nonlinear models, in either the time domain or the frequency domain.

For working in the time domain, the symbolically defined device (refer to *Custom Modeling with Symbolically-Defined Devices* (modbuild)) enables users to specify nonlinear models directly on the circuit schematic, using algebraic relationships for the port voltages and currents. It works very well for creating many nonlinear device models, but it can be cumbersome for describing the nonlinear, behavioral, frequency-domain operation of the type of subsystems used in RF and microwave communication systems.

To address this need, the *frequency-domain defined device* (FDD) was developed. The FDD enables you to directly describe current and voltage spectral values in terms of algebraic relationships of other voltage and current spectral values. This simplifies development of non-linear, behavioral models that are defined in the frequency domain. The FDD is ideal for modeling a variety of devices, such as modulators and demodulators, phase lock loop components, and more.

The FDD includes capabilities that make it well suited for modeling digital communication subsystems, which often behave in ways that cannot be adequately modeled as time-invariant. Clocked systems, sampled systems, TDMA pulsed systems, and digitally-controlled systems are common, even in the RF and microwave area, and behavioral models must be able to include these effects. So, in addition to its frequency-domain modeling attributes, the FDD also enables you to define trigger events, to sample the voltages and currents at trigger events, and to generate outputs that are arbitrary functions of either the time of the trigger or of the complex spectral voltage and current values at these trigger events.

While the SDD is the user-defined model of choice for modeling at the device and component level where physics dictates that responses are a function of the instantaneous port variables, the FDD is preferable for nonlinear, behavioral modeling in both the frequency and time domains.

Before continuing this section, you should be familiar with the SDD. This section assumes knowledge of several topics that are presented in the discussion of SDDs, such as port variables and explicit and implicit equations. For more information, refer to *Custom Modeling with Symbolically-Defined Devices* (modbuild).

Signal Models and Sources

To fully understand how FDD models work and what they can do, some understanding

of how the simulator models signals in the different simulation analyses is necessary. While the descriptions that follow use voltages, either voltage or current signals can be used.

In *DC* analyses, a node voltage is simply expressed as a constant V for all time. Its frequency spectrum is simply an impulse at DC with a value of V .

In *transient* and *convolution* analyses, a node voltage is still a single variable, but it is now a time-varying variable $V(t)$, which can theoretically represent any type of signal from DC up to the Nyquist bandwidth ($.5/T_{\text{step}}$). These signals can be periodic, transient, or random signals. The spectrum of this signal can be computed with Fourier transform techniques.

In *harmonic balance* analyses, a node voltage is represented by a discrete spectrum in the frequency domain. This limits the signal types to quasi-periodic signals, and, given memory limitations, to a relatively few number of discrete frequencies. The time-domain waveform can be computed by Fourier transform techniques, based on the equation below.

$$V(t) = \text{real} \left(\sum_{k=0}^N V_k e^{j2\pi f_k t} \right)$$

The set of harmonic frequencies is defined by the user entering a set of fundamental frequencies, with an order for each tone. A maximum order parameter is also required for limiting the number of mixing tones that are included in the set of harmonic frequencies. For each of these frequencies, each node voltage has a constant value associated with it, signifying the amplitude and phase of the periodic sinusoid at that frequency.

These frequencies are referenced by fundamental frequency indices, in the following manner:

- Given the indices $[m, n]$, the corresponding frequency is $m \cdot \text{freq1} + n \cdot \text{freq2}$, where freq1 and freq2 are fundamental frequencies.

For example, consider a two-fundamental simulation, with fundamental frequencies freq1 and freq2 defined as 1GHz and 900 MHz, respectively. The frequency component at 1 GHz would have indices of $[1,0]$. The 900 MHz frequency component would have indices of $[0,1]$. 100 MHz would have an index of $[1,-1]$, and $[2,-1]$ would be one of the intermod terms at 1.1 GHz. Note that $[0,0]$ refers to DC. Indices of $[-1,1]$ reference -100 MHz and its spectral values would be equal to the complex conjugate of those at 100 MHz.

A three-fundamental frequency system requires three indices $[m, n, o]$ to define a unique frequency component.

In *Circuit Envelope* analyses, a node voltage is represented by a time-varying, frequency-domain spectrum. As in harmonic balance, a set of harmonic frequencies is user-defined. But here, the spectral amplitude and phase at each of these frequencies can vary with time, so the signal it represents is no longer limited to a constant sinusoid. Each of these harmonic frequencies is the center frequency of a spectrum; the width of each spectrum is

$\pm 0.5/T_{\text{step}}$. The bandlimited signal within each of these spectra can contain multiple periodic, transient, or random tones. The actual time-domain waveform is now represented by the following equation.

$$V(t) = \text{real} \left(\sum_{k=0}^N V_k(t) e^{j2\pi f_k t} \right)$$

Since each time-varying spectrum $V_k(t)$ can be thought of as a modulation waveform of the center frequency f_k , these are often referred to as *envelopes*. This does not imply that there must actually be a frequency component at the center frequency, see the following table for examples. Note there are $N+1$ of these spectra. The one at DC (also referred to as the baseband component) is limited to a bandwidth of $0.5/T_{\text{step}}$ and must always be real. The other N spectra have a double-sided bandwidth of $1/T_{\text{step}}$ and are usually complex.

Example Signals for Spectrum around f_k

#	Formula	Description
1	$V_k=1$	Constant cosine $\cos(2\pi f_k \text{time})$
2	$V_k=\exp(-j\pi/2)$ or $\text{polar}(1,-90)$ or $-j$	Constant sine $\sin(2\pi f_k \text{time})$
3	$V_k=A \exp(j(2\pi f_m \text{time} + B))$	One tone (SSB) $A \cos(2\pi (f_k + f_m) \text{time} + B)$
4	$V_k=A \exp(jB)$; $\text{freq}=1.1 \text{ GHz}$ ¹	Same as (3) (assuming $f_k + f_m = 1.1 \text{ GHz}$)
5	$V_k=2 \cos(2\pi f_m \text{time})$	Two tone (AM suppressed carrier)
6	$V_k=\exp(j2\pi f_m \text{time}) + \exp(-j2\pi f_m \text{time})$	Same as (5)
7	$V_k=\text{pulse}(\text{time}, \dots)$; $\text{freq}=f_k + f_m$ ¹	Pulsed RF at a frequency of $f_k + f_m$
8	$V_k = -\text{step}(\text{time} - \text{delay})$	A negative cosine wave, gated on at $t=\text{delay}$
9	$V_k = (\text{vreal}(\text{time}) + j \text{vimag}(\text{time})) \exp(j2\pi f_m \text{time})$	I/Q modulated source centered at $f_k + f_m$. ($\text{vreal}()$, $\text{vimag}()$ user-defined functions)
10	$V_k = (1 + \text{vr1}) \exp(j2\pi \text{vr2})$	Amplitude and noise modulated source at f_k . (vr1 , vr2 user-defined randtime functions)
11	$V_k = \exp(j2\pi (-f_0 + a_0 \text{time}/2) \text{time})$	Chirped FM signal starting at $f_k - f_0$, rate a_0

1. _freq is defined in the paragraph below.

The envelope waveform $V_k(t)$ has many useful properties. For example, to find the instantaneous amplitude of the spectrum around f_k at time t_s , you simply compute the magnitude of complex number $V_k(t_s)$. Similarly, the phase, real, and imaginary values of instantaneous modulation can be extracted by simply computing the phase, real, and imaginary values of $V_k(t_s)$. Note this is only extracting the magnitude of the modulation around f_k . It is not including any of the spectral components of adjacent f_{k-1} or f_{k+1} spectra, even if these spectra actually overlap. If this f_k spectrum has multiple tones inside of it, then this demodulation does include their effects.

This simple technique does not allow demodulating only one of the tones inside this f_k spectrum and excluding the other tones in the f_k spectrum. To accomplish this, the desired tone must first be selected by using an appropriate filter in the circuit simulation. Also note that since the baseband (DC) spectrum represents a real signal and not a complex envelope, its magnitude corresponds to taking the absolute value, and its phase is 0 or 180 degrees.

Defining Sources

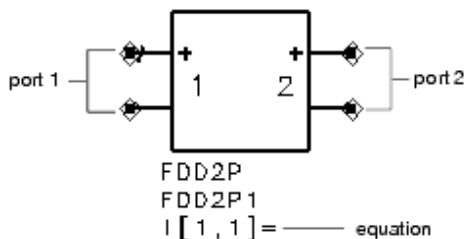
To define a source for Circuit Envelope, you first identify in which spectral envelope the signal belongs. This will typically be the fundamental of one of the frequencies specified in the analysis. Most frequency-domain sources have a single frequency parameter that can be specified. When these sources are used in a harmonic balance or Circuit Envelope simulation, the simulator will determine which of the analysis frequencies is the closest to the source frequency, and if it is close enough, will internally assign it the corresponding set of indices. A Circuit Envelope simulation will also determine the frequency offset from the analysis frequency and automatically shift the signal accordingly. This frequency offset can be up to $\pm 0.5/\text{Timestep}$. If the source frequency is too far away from any analysis frequency, then its output is set to 0.0 for that analysis and a warning is generated.

Regarding the equations used to define an output from these sources, instead of having to define a fundamental frequency and an SSB frequency offset modulation as in source example 3 in [Example Signals for Spectrum around \$f_k\$](#) , the simpler format of example 4 is now possible. In addition, these frequency-defined sources are also directly compatible with simple transient analysis.

The Frequency-Domain Defined Device

This section describes the equations and parameters of the FDD. A procedure for adding an FDD to a schematic is in the section [Adding an FDD to a Schematic](#). For examples of FDDs developed into models, refer to the section [FDD Examples](#).

The frequency-domain defined device is represented on the circuit schematic as an n-port, along with equations specifying the relationships between the spectral port variables. An example of a 2-port FDD is shown here.



By usual convention, a positive port current flows into the terminal marked +.

Retrieving Values from Port Variables

The variables of interest at a given port are the port spectral voltages and port spectral currents. Spectral voltages and currents can be obtained using the functions `_sv()`, `_si()`, `_sv_d()`, and `_si_d()`, which are described in the following table, and used in conjunction with equations, which are described in the section [Defining Constitutive Relationships with Equations](#). The `_sv()` and `_si()` functions return voltage or current values for a specific port and for a specific frequency. You choose the port by port number, and you choose the frequency using a frequency index. The index is either:

- The index to an FDD carrier frequency and its harmonics
- A set of indices that reference the frequencies of a harmonic balance analysis

For information on FDD carrier frequencies and indexing, refer to the section [Specifying Carriers with the Freq Parameter](#). For a description of frequency indices from a harmonic balance analysis, refer to the section [Signal Models and Sources](#).

As an example, to access the spectral voltage at port 1 for the second harmonic of the first fundamental frequency, use the function `_sv(1,2)`. Similarly, to access the port n current at the frequency with indices $[h1,h2,h3]$, use the function `_si(n , $h1$, $h2$, $h3$)`. Both of these functions return single complex values at each time point, unless the specified envelope is baseband, in which case the value is real. The underscore in the function names is used so as to not conflict with user-defined functions, and to signify that these functions only have meaning within the context of evaluating the FDD device. They cannot be used to directly generate output variables, for example.

The `_sv()` and `_si()` functions return the present value of the specified spectral envelope. For transient, convolution, or Circuit Envelope simulations, it is also desirable to access past values of the spectral port variables. This can be done using the `_sv_d()` and `_si_d()` functions, which are described in the following table. These functions have a delay parameter. For example, to find the value of the port 2 voltage at the $[2,-1]$ intermod frequency 10 μ sec ago, use `_sv_d(2, 10us, 2,- 1)`.

Functions for Accessing Spectral Ports and Currents

Name	Description
<code>_sv(port, index)¹</code>	Returns the spectral voltage for the specified port at the specified frequency index.
<code>_si(port, index)</code>	Returns the spectral current for the specified port at the specified frequency index.
<code>_sv_d(port, delay, index)</code>	Returns a past value of spectral voltage for the specified port and time delay at the specified frequency index.
<code>_si_d(port, delay, index)</code>	Returns a past value of spectral current for the specified port and time delay at the specified frequency index.

¹ index can refer either to the index of an FDD carrier frequency as defined with the freq parameter and its harmonics or a set of indices that reference the fundamental frequencies of a harmonic balance analysis.

The delay values in the `_sv_d()` and `_si_d()` functions can be variables that change during the simulation. However, these delay variables must have their maximum value at time $t=0$. This is to allow proper initialization of the required history buffers. This criteria can usually be met with an expression such as

fdd_delay = if (time = 0) then max_delay else variable_delay endif.

where *max_delay* is some reasonable value that the *variable_delay* is known to never exceed.

Also, FDDs require that all state variable dependencies that will ever exist must exist at time = 0. For example, the following equation describes a discrete accumulation operation, with a reset to 0 at time = 0:

v[2,0] = if (time = 0) then 0.0 else _sv_d(2,timestep) + _sv(1,0) endif

However, it must be modified to work with the FDD so that both state variable dependencies as well as the maximum delay at time = 0. The following satisfies this criteria by adding an insignificant portion to the time = 0 value.

next_state = _sv_d(2,timestep) + _sv(1,0)

*v[2,0] = if (time = 0) then 0.0 + next_state*1.0e-100 else next_state endif*

Defining Constitutive Relationships with Equations

An unlimited number of equations can be used to define constitutive relationships between the port voltages and port currents. There are two basic types of equations allowed, current equations and voltage equations. Their basic format is:

i[port, findex] = f(_sv(),_sv_d(),_si(),_si_d())

v[port, findex] = f(_sv(),_sv_d(),_si(),_si_d())

The equations can be listed in any order, and more than one equation can be used for a single port, but each port must have at least one equation.

Note the use of indices on the left side of the equations. This is similar to the use of indices in the *_sv()* and other functions that were previously described, they can be either the index to an FDD carrier frequency and its harmonics or a set of indices that reference the frequencies of a harmonic balance analysis. Indices are discussed in the sections [Signal Models and Sources](#) and [Specifying Carriers with the Freq Parameter](#).

In order for a port current to be used on the right side of an equation, at least one voltage equation for that port must be defined. It does not matter which harmonic indices are used for this. Normally, the simulator does not generate current-state variables, only node voltage-state variables. This is sufficient as long as the devices are modeled as voltage-controlled current sources. This is also the most efficient method in terms of speed and memory. However, current-controlled sources can be generated but the simulator requires an additional equation to define this current. The presence of a voltage equation signifies this to the simulator. In general, the voltage equations should only be used when the voltage-controlled current equations are insufficient. Refer to *Modified Nodal Analysis* (modbuild), for more detail.

While the SDD has truly implicit equations of the form $f(_v1,...,_vn,_i1,...,_in)=0$ the FDD does not. However, the equivalent effect can be generated by adding the left side to the right side of the voltage equations. For example,

$$v[2,1,0]=f(_sv(1,0),_si(2,1,0))+_sv(2,1,0)$$

has effectively generated the implicit equation $f(...)=0$. The FDD is different in this respect in order to solve the problem of how to define the voltage at all unspecified spectral frequencies. The above voltage equation also generated a number of additional implied equations that say

$$v[2,all\ other\ harmonic\ indices] = 0.0.$$

Note that all equations for the same spectral port variable are effectively added together, so if you also define an equation for another spectral frequency at the same port (i.e., $v[2,0] = vcalc$), this additional implied equation that sets the voltage to zero is ignored.

For a procedure on how to add current and voltage equations to an FDD, refer to the section [Defining Current and Voltage Equations](#).

Continuity

As with any Newton-Raphson based circuit-solving algorithm, the constitutive relationships should be differentiable with respect to each of the specified spectral voltages and currents, and these derivatives should be continuous. Discontinuous derivatives may cause the simulator to have trouble converging.

One possible technique to improve convergence, or to circumvent the above limitations, is to add delay between the input and output using the `_sv_d()` or `_si_d()` functions. If this delay is greater than the simulation timestep, then the derivative information is no longer needed or used. If this delay is acceptable behavior for the model, simulation speed can be improved.

Large step discontinuities in the time-domain functions can also create convergence problems, either taking longer to solve or possibly causing convergence failure. Although having continuous derivatives with respect to time is not as important as having continuous derivatives with respect to the spectral port variables, care should be taken when using abrupt time functions.

Specifying Carriers with the Freq Parameter

The FDD has a repeatable `freq[n]` parameter, which can be used to define one or more carrier frequencies. The `[n]` is used to identify each carrier, for example, `freq[1]=100 MHz`, `freq[2]= 350 MHz`, `freq[3]=800 MHz`. If carrier frequencies are defined for the FDD, you can reference them using `_sv()` and related functions in order to collect voltages and currents at carrier frequencies and their harmonics.

The syntax to reference these carrier frequencies defined in the FDD components is to set the first index parameter equal to the negative of the carrier frequency number $[n]$. The optional second index parameter specifies the harmonic of that carrier frequency. If nothing is specified for the second index parameter, then a default of 1 is used. For example, if an FDD has defined $freq[3]=800\text{ MHz}$, then $sv(1,-3,3)$ specifies the port 1 spectral voltage at the envelope closest to 2.4GHz. Here -3, indicates that $freq[3]$ defined in the FDD component will be used and 3 indicates that the 3rd harmonic of the $freq[3]$ value is desired. Thus $3 * freq[3] = 2.4\text{ GHz}$. If there is no analysis frequency close enough to 2.4 GHz (within 0.5/Timestep for Circuit Envelope), then this function simply returns 0.0 and generates a warning.

Also, if an FDD has defined $freq[3]=800\text{ MHz}$, then $sv(1,-3)$ specifies the port 1 spectral voltage at the envelope closest to 800 MHz. Here, -3 indicates that $freq[3]$ defined in the FDD component will be used. And since nothing is specified for the second index parameter, 1 can be used. Thus $1 * freq[3] = 800\text{ MHz}$.

Note that $sv(1,3,3)$ would be unaffected by the carrier frequency parameters ($freq[n]$) defined in the FDD component since the first index number is positive (3). However, it would still refer to the envelope at $3 * freq1 + 3 * freq2$, where $freq1$ to $freq12$ are predefined variables that are set to the fundamental frequencies defined by the analysis.

The FDD $freq$ parameter behaves differently than the source $freq$ parameter (referred to in the section [Defining Sources](#)) in that any acceptable offset frequency between the carrier frequency and envelope center frequency is *ignored*. For example, given an envelope analysis with a fundamental frequency of 0.5 GHz, a timestep of 1μsec, and FDD $freq[1] = 500\text{MHz}$ and $freq[2]=500.1\text{MHz}$, then $sv(1,-1)$ and $sv(1,-2)$ would return the same value, the port 1 spectral voltage at 500 MHz. (If timestep is changed to 1μsec, then $sv(1,-2)$ would return 0.0.)

Note that it is not possible to reference a mixing product of multiple carrier frequencies. If this is desired, then an additional carrier frequency equal to the desired mixing product frequency must be defined. For an example, refer to the mixer example in the section [Mixer](#).

For a procedure on how to add $freq$ parameters to an FDD, refer to the section [Defining Frequency Parameters](#).

Creating Output Harmonics

If you are creating a model with the capability to output a large number of harmonics, the $harm$ variable can be used with the FDD to develop a such a model. The $harm$ variable, unlike most of the functions described in this section, is not restricted to use in FDDs only.

Using the $harm$ variable, voltage and current source harmonic values can be parametrically defined. Anytime the large signal voltage or current is defined with an expression using $harm$, the device will automatically index the value of $harm$ from 0 to the maximum value needed for the present analysis. The expression is then re-evaluated

for each value of `_harm` to determine the spectral content at a frequency equal to `_harm*frequency` determined by the parameter indices.

The variable `_harm` is used in the function below, which implements a pulse source:

```
parameters VPEAK=1 V DUTYCYCLE=30 FREQ=1 GHz HARMONICS=16
ivs: CMP79 1 2 freq=Freq v[1]=A(_harm)
Blackman(n,M) = (0.42 + 0.50*cos(PI*n/M) + 0.08*cos(2*PI*n/M))*step(M-n)
A(k)=2*VPEAK*DUTYCYCLE/100*sinc(k*pi*DUTYCYCLE/100)*(-1)^k
* Blackman(k,HARMONICS+1)*step(_harm)
```

The variable `_harm` has no set maximum harmonic limitation. The value of `_harm` will automatically be incremented out to the maximum value available for the present analysis.

This automatic indexing also works in baseband envelope and transient. The variable is incremented until it either reaches 1000 or until its amplitude has become insignificant for several consecutive harmonics.

In non-baseband envelope, the maximum harmonic will also be limited if the source's harmonic falls outside the envelope bandwidth. For example, if the analysis fundamental is 1MHz with a timestep of 1µsec (+/-500Hz envelope bandwidth) and the source fundamental frequency is 1MHz + 100Hz, then the 6th harmonic falls outside the envelope bandwidth and the spectrum is truncated, even if the analysis order is 31. Also, anytime a spectrum is truncated in harmonic balance, it remains truncated even if higher order spectral tones may exist, for example, if another fundamental existed at 10MHz+1kHz, the spectral source would not add energy there even though it is at the 10th harmonic of the source.

Note that the frequency value is determined by the frequency defined in the parameter indices. In the above case, for example, `v[1,2] = A(_harm)` would have defined a pulse waveform whose fundamental frequency is the second harmonic of `Freq`. The equation `v[0,1,2] = A(_harm)` will define a waveform whose fundamental frequency is $(1 * \text{freq1} + 2 * \text{freq2})$ where the `_freqN` variables are the fundamental frequencies defined by the analysis. At the netlist level, multiple different spectrum can be defined in one source, but each one will add the DC (`_harm=0`) value.

Limitations

In general, you should avoid using a fundamental frequency of 0 Hz. The `_harm` parameter is not supported for the small signal spectral parameters.

There is no simulator variable available to determine what the maximum number of harmonics is for a particular case. This can make windowing a little difficult, since a parameter must be used or passed to the model to set the window bandwidth.

Defining an FDD Spectrum

The parametric definition of output spectrum using the `_harm` index also works with the FDD. This example defines a VCO:

```
parameters Kv=1khz FUND=1 Rout=50 ohm PFUND=0 dBm Harmonic=
fdd:COMP2 _NET00005 0 out 0 i[1,0]=0
i[2,f1,f2,f3]=a*Harmonic*exp(j*_harm*b)*step
(_harm)
a = -dbmtoa(PFUND, a_Rout)
a_Rout = max(Rout,0.1)
b = 1000*_sv_d(1,timestep,0)-pi/2
f1 = if (FUND = 1) then 1 else 0 endif
f2 = if (FUND = 2) then 1 else 0 endif
f3 = if (FUND = 3) then 1 else 0 endif
C:COMP3 0 _NET00005 C=1/(2*pi*Kv)
vco_harm:COMP1 vin vout Kv=1khz FUND=1 Rout=50 ohm PFUND=0 dBm
Harmonic=fharm
fharm = sinc(_harm*pi*.25)/sinc(pi*.25)
```

In this example, the user has entered the equation for the spectrum of a 25% duty cycle square wave using the `_harm` index, which can generate as many harmonics as can be supported by the present analysis. The fundamental power is still separately defined so this spectrum is relative to that and the value for `_harm=1` should be 1.0.

This method of harmonic indexing in the FDD is meant primarily for defining multiple spectral outputs dependent on the same spectral input. But the `_harm` index can also be used to change which spectral input is used for each spectral output. An example is `i[2,1]=_sv(2,_harm)/50`, which adds a 50 ohm load at all the harmonics of `fund1`, including DC. Note that this example is to illustrate the capability, it is inefficient compared to using a resistor. Another example is `i[2,1] = _sv(1,0,_harm)/mag(_sv(1,0,1)+tinyreal)/Rout`, which outputs an entire `fund1` spectrum at port 2, based on the port 1 `fund2` spectrum, and limits each spectral component by the `fund2` fundamental magnitude.

Note there is no capability in the FDD to allow automatic outputting of all spectral tones. The `_harm` index is essentially limited to harmonics of the frequency specified by the parameter indices. Additional parameters and equations have to be used to cover additional fundamentals and intermods. An example of this is in the section [Mixer](#).

Using Arrays

Sometimes you may want a flexible number of spectral tones, but no simple equation is available for this. Arrays can be used in this case, and unlike the `harmlist` parameter, a separate parameter with separate supporting code is not required. Care must be taken to avoid having the indexes exceed the array bounds, or an error will occur. For this reason, a `length()` function is available to return the length of an array. The following VCO example shows a possible usage. Also, note that the `Harmonics` input parameter is a list of complex numbers representing the relative level of all the desired harmonics, for example, `list(.1, .02_j*.01)`.


```

parameters Kv=1khz Freq=1GHz P=-j*dbmtow(0) Rout=50 Ohm
Delay=timestep Harmonics=
; Y_Port is used as voltage to current converter
_Y_Port: CMP1 in 0 NET00005 0 Y[2,1]=-0.001
; This capacitor performs the integration function
C: CMP3 0 _NET00005 C=1/(2*pi*Kv)
; This switch resets the integrating capacitor voltage to 0 at time = 0
ResetSwitch: CMP4 0 _NET00005
; This FDD is a programmable harmonic current source with a phase modulation
input
FDD: CMP2 _NET00005 0 out 0 I[1,0]=0 I[2,-1]=a*exp(j*b)
I[2,-1]=if (_harm > 1 and _harm <= Hmax+1) then a*Harmonics[_harm-
1]*exp(j*_harm*b)
else 0.0 endif Freq[1]=Freq
Hmax = length(Harmonics)
Pdbm=30.0 + 10.0*log(mag(P) + tinyreal)
a = -dbmtoa(Pdbm, Rout)*exp(j*phaserad(P))
b = 1000*_sv_d(1,Delay,0)
R: CMP5 out 0 R=Rout

```

In this case, the model was hard-coded to expect an array. A more general solution might use the above VCO example, but this would require the user to limit the array bounds access, since accessing out of bounds will cause an error.

```

vcodata=makearray(0,1,2,.5,.25*j,.125,-.0625,-j*.03125,.015625)
xyz = if (_harm < length(vcodata) then vcodata[_harm] else 0.0 endif
vco_harm: CMP1 vin vout Kv=1khz FUND=1 Rout=50 ohm PFUND=0 dBm
Harmonic=xyz

```

While this could be used to simplify the *harmlist* implementation, *harmlist* may be more efficient.

Trigger Events

The FDD enables you to define trigger events. Up to 31 triggers can be defined. Anytime the value of the trigger expression is equal to a number other than zero, a trigger event is declared for the corresponding trigger. Each trigger keeps a count of the number times the trigger occurred and the time of its last trigger. The trigger time is defined as the time value of the current simulation point plus the value of the expression. Therefore the value of the expression should normally be the time of the trigger relative to the current time value. The value of this trigger expression should be limited to *-timestep* and *-2*timestep*. This is explained further in the section [Accessing Port Variables at Trigger Events](#).

Three built-in functions have been defined to provide access to trigger information, they are described in the following table. Again, the underscore is used as part of the name, signifying that these functions only have meaning within the context of an FDD instance, and are not valid elsewhere.

Functions available to access trigger information

Name	Description
<code>_to(N)</code>	Returns 1 if trigger N occurred at this time point, else 0
<code>_tn(N)</code>	Returns the accumulated number of trigger N events
<code>_tt(N)</code>	Returns the absolute time in seconds of last trigger N event

Another function is available to detect threshold crossings and to generate the proper trigger expression values, which is shown in *Function to generate a trigger event*. Note that the threshold crossings are based only on the DC (baseband) spectral voltage at the specified port. The actual time crossing is computed based on linear interpolation between adjacent time points, so the actual accuracy will depend on both the size of the time step and the rate of change of the slope of the signal.

Function to generate a trigger event

Another function is available to detect threshold crossings and to generate the proper trigger expression values, which is shown in *to generate a trigger event*. Note that the threshold crossings are based only on the DC (baseband) spectral voltage at the specified port. The actual time crossing is computed based on linear interpolation between adjacent time points, so the actual accuracy will depend on both the size of the time step and the rate of change of the slope of the signal.

Name	Description
<code>_xcross(P, Vthresh, direction)</code>	Returns 0 if no threshold crossing occurred, otherwise returns its relative time, a value between (-1 and -2)*timestep. A threshold crossing occurs if the baseband voltage at port P passes through the value Vthresh in the specified direction. A positive direction number implies a positive edge; a negated number a negative edge; a direction number of 0 implies either positive or negative edge. No hysteresis exists.

For a procedure on how to add trigger parameters to an FDD, refer to the section [Defining Triggers](#).

Output Clock Enables

Normally all of the FDD voltages and currents are re-evaluated at every time sample. It is possible, though, to enable the output of a given port to change only when a specified trigger, or a set of specified triggers, occurs. This is done using the clock enable parameter, `ce[n]=value`. `[n]` specifies the port where the clock enable will be applied. `value` is a binary value that is set using the `bin()` function, where the Nth bit corresponds to whether this port should be enabled by the Nth trigger. For example, if you want the output of port `n` to be updated whenever either trigger 1 or trigger 3 occur, you would enter a value of `bin(101)` or 5 for the clock enable parameter. Clock enables can be used when it is necessary to update computed values only at certain time points. Sample-and-holds are one obvious application, refer to the example in the section [Sample and Hold](#).

For a procedure on how to add clock enable parameters to an FDD, refer to the section [Defining Clock Enables](#).

Accessing Port Variables at Trigger Events

Now that it is possible to generate trigger events at threshold crossings, it is desirable to be able to determine the spectral port voltages and currents at the point in time that this trigger occurred. Linear magnitude and phase interpolation is used to compute values at times between adjacent simulator time points, and again, the accuracy depends on the rate of change of the input envelope waveform. The four functions that are used to do this are described in *Functions to access port variables at trigger events*.

Functions to access port variables at trigger events

Name	Description
<code>_sv_e(P,N,indices)</code>	Return the port P spectral voltage envelope at the last trigger N time
<code>_si_e(P,N,indices)</code>	Return the port P spectral current envelope at the last trigger N time
<code>_sv_bb(P,N)</code>	Return the total, real voltage of port P at the last trigger N time
<code>_si_bb(P,N)</code>	Return the total, real current entering port P at the last trigger N time

The `_sv_e()` function is very similar to `_sv_d()`, which is described in [*Functions for Accessing Spectral Ports and Currents* | Custom Modeling with Frequency-Domain Defined Devices#1104151]. By default, though, past history for the `_sv_e()` function is only saved for the last 2 timesteps. Therefore, the event they refer to must have just occurred, and cannot delay back an arbitrary amount of time. If a triggered voltage value is desired at a much later point in time, then it should be sampled and held using a combination of the above functions and the clock enable previously discussed.

All of the spectral port variable functions discussed so far return only the complex value of the single specified envelope. (If the indices are 0, then the real baseband value is returned.) The broadband functions `_sv_bb()` and `_si_bb()` functions, though, perform an inverse Fourier transform of all of the spectral voltages or currents at the specified event time, and return the real value. Note that if this value is computed at every time step, it will generate an aliased, undersampled waveform, since the time step in Circuit Envelope is typically much less than the period of the various envelope center frequencies.

Delaying the Carrier and the Envelope

With exception of the `_sv_bb()` and `_si_bb()` functions, all of the other spectral port variable functions return the envelope information. This is true even with the delayed and event versions. If it is necessary to delay both the envelope and the carrier, then an additional term must be added to account for the carrier phase shift. For example, if the fundamental signal is

$$V_k(t) * \exp(j * 2 * \pi * f_c * t)$$

then

$$i[2,1] = _sv_d(1,1\text{usec},1)$$

generates a current equal to

$$V_k(t-1 \text{ m sec}) * \exp(j * 2 * \pi * f_c * t)$$

To generate a true coherent delay with the FDD, you would have to modify the equation to

$$i[2,1] = _sv_d(1,1\text{usec},1) * \exp(-j * 2 * \pi * f_c * 1\text{usec})$$

or to something similar. Of course, if only a fixed delay is desired, there are linear elements that are more suitable for this application than the FDD.

Miscellaneous FDD Functions

There are three remaining functions that are available in the FDD for time-domain operations, and are described in *Miscellaneous FDD Functions*. They were incorporated into the FDD because they required that state history be maintained. The functions correspond to a basic counter and to a linear feedback shift register. These functions are valid only when used in an FDD.

Name	Description
<code>_divn(T,N,N0),</code>	Returns the value of a counter, clocked every time trigger T occurs, decrementing from N to 0 . $N0$ is initial time = 0 value.
<code>_lfsr(T, seed,taps)</code>	Returns the value of a linear feed back shift register that is clocked every trigger T . $seed$ is the initial value of the register. $taps$ are the binary weights of the bits that are fed back using modulo 2 math.
<code>_shift_reg(T,M, N, In)</code>	Returns the value of a multi-mode shift register that is clocked every trigger T , has N bits, and with an input equal to In . $M = 0$: LSB first, Serial In, Parallel Out $M = 1$: MSB first, Serial In, Parallel Out $M = 2$: LSB first, Parallel In, Serial Out $M = 3$: MSB first, Parallel In, Serial Out

Defining Input and Output Impedances

With the SDD, it is very straight-forward to include the input and output resistances in the basic equations. For example, $i[1] = _v1/50$ simply defines a 50 ohm input resistance. This is not as simple with the FDD, since each equation only defines the relationship for a single output spectrum. Thus, $i[1,1,0] = _sv(1,1,0)/50$ defines a 50 ohm input resistance, but only for the fundamental spectral envelope. The input resistance for the other spectral components is still infinite, which is the equivalent of being undefined. This becomes more problematic at the output. It is possible to define the output current and output resistance for a single spectral envelope, but to leave the other spectral envelopes undefined. This may create an ill-defined circuit, creating a singular matrix error due to an undefined voltage at certain spectral frequencies. These problems are best circumvented by using actual resistors external to the FDD. Of course, if the resistance for certain spectral envelopes is different from this external value, that difference can be included in the defining spectral port equations.

Compatibility with Different Simulation Modes

The FDD is not fully compatible with all the different circuit analysis modes of Advanced Design System. Since DC, AC, transient, and convolution analyses only define the baseband variables, any use of non-baseband spectral envelopes (harmonic indices not equal to 0) are ignored in these analyses and the voltages and currents for these spectral frequencies are set to 0.

Similarly, DC, AC, and harmonic balance analyses are steady-state analyses and time is always equal to 0, so any time-varying functions are evaluated at time=0 and accessing delayed voltages is the same as accessing the present voltage. The concept of generating time trigger events, of course, is valid only in transient, convolution, and Circuit Envelope modes of operation.

Components Based on the FDD

A variety of circuit components in Advanced Design System are based on the FDD. Some of these components are:

- Tuned modulators and demodulators
- Phase lock loop components
- Counter, time, and waveform statistics probes
- Sampler

Many of these models operate on a few (often just one) of the input spectral frequencies, and in turn output just one, or a few, different spectral frequencies. This is consistent with the desired, or measured, primary frequency-domain behavior, and simulations can be performed quite efficiently since all operations are done directly in the frequency domain.

In cases where a model must include second and third-order interactions with other spectral frequency components, and the underlying nonlinearity is an algebraic function of the time-domain voltages and currents, the FDD may become too tedious to generate all of the frequency-domain equations that define the multiple interactions, and a broadband model (which can be developed using the SDD) may be the preferred model.

The FDD spectral models, in general, will not function with AC and transient analyses. These limitations are noted where the components are documented in *Introduction to Circuit Components* (ccsim).

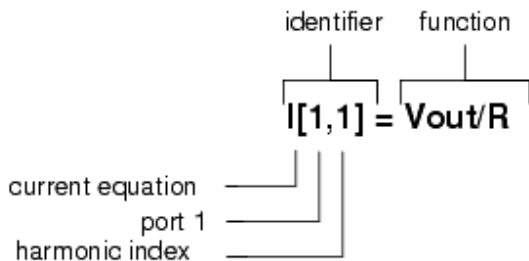
Adding an FDD to a Schematic

FDDs can be added to a schematic in the same way as other components are added and connected to a circuit. This section describes the mechanics of adding an FDD component to a schematic and defining it.

To add an FDD:

1. From the Component Palette List, choose **Eqn-based Nonlinear**.

2. Select the FDD with the desired number of ports, add it to the schematic, and return to select mode.
3. Double-click the FDD symbol to edit the component.
4. FDD parameters are entered in the Select Parameters list. The parameter is on the left side of the equation. It identifies the type of parameter, the port it is applied to, and, where appropriate, the harmonic index. Select the parameter you want to edit. (Note the buttons below the list to add, cut, and paste parameters as necessary.)



5. Under Parameter Entry Mode, specify the type of parameter to be defined: **current**, **voltage**, **frequency**, **trigger**, or **clock enable**. Instructions for defining each type of parameter follow.
6. Once a parameter is defined, click **Apply** to update.
7. Add and edit other parameters as desired.
8. Click **OK** to accept the changes and dismiss the dialog box.

Defining Current and Voltage Equations

Current and voltage equations are the two basic types of equations for defining constitutive relationships between the port voltages and port currents. For more information about these equations, refer to the section [Defining Constitutive Relationships with Equations](#).

To define current or voltage equations:

1. Double-click the FDD component to open the Edit Component dialog box.
2. By default, a current equation appears in the Select Parameters list. Select this equation.
3. From the Parameter Entry Mode list, choose either **Current** or **Voltage**. For current equations, an I appears on the left side of the equation; for voltage equations, a V is displayed.
4. In the Port field, enter the number of the port that you want the equation to apply to.
5. In the Harmonic indices field, enter the harmonic index that the equation applies to, either an absolute index, or a locally-defined carrier frequency, in which case the first index must be negative.
6. In the Formula field, enter the expression that defines the current or voltage.
7. Click **Apply** to update the equation.
8. To add another equation, click **Add** and repeat steps 3-7.
9. Click **OK** to accept the changes and dismiss the dialog box.

Defining Frequency Parameters

The *freq* parameter can be used to define one or more carriers for an FDD. For more information about the *freq* parameter, refer to the section [Specifying Carriers with the Freq Parameter](#).

To define a frequency parameter:

1. Double-click the FDD component to open the Edit Component dialog box.
2. Select any parameter in the Select Parameters list.
3. Click **Add**. The new parameter is automatically selected.
4. From the Parameter Entry Mode list, choose **Frequency**. The left side of the equation is changed to **Freq[n]**, where *n* is an index indicating that it is the *n*th frequency parameter defined for the FDD.
5. In the Index field, enter the index that identifies the frequency.



Note

This index is used only to specify which frequency parameter to use when more than one envelope is specified for an FDD. It does not specify a frequency offset.

6. In the Formula field, enter the expression that defines the frequency.
7. Click **Apply** to update the parameter.
8. Click **OK** to accept the changes and dismiss the dialog box.

Defining Triggers

Up to 31 triggers can be defined for a single FDD. Any time the value of the trigger expression is equal to a value other than zero, a trigger event is declared for that trigger. Each trigger keeps a count of the number of times the trigger occurred and the time of the last trigger. For more information about triggers, refer to the section [Trigger Events](#).

To define a trigger:

1. Double-click the FDD component to open the Edit Component dialog box.
2. Select any equation in the Select Parameters list.
3. Click **Add**. The new equation is automatically selected.
4. From the Parameter Entry Mode list, choose **Trigger**. The left side of the equation is changed to **Trig[n]**, where *n* identifies the trigger.
5. In the Index field, enter the value that identifies the trigger, 1-31.
6. In the Formula field, enter the expression that defines the trigger event.
7. Click **Apply** to update the parameter.
8. Click **OK** to accept the changes and dismiss the dialog box.

Defining Clock Enables

Clock enables restrict FDD voltages and currents to change only when a specified trigger, or a set of specified triggers, occurs. This is done by setting the clock enable of the desired port to a binary value, where the *N*th bit corresponds to whether this port should

be enabled by the Nth trigger. For more information, refer to the section [Output Clock Enables](#).

To define a clock enable:

1. Double-click the FDD component to open the Edit Component dialog box.
2. Select any equation in the Select Parameters list.
3. Click **Add**. The new equation is automatically selected.
4. From the Parameter Entry Mode list, choose **Clock Enable**. The left side of the equation is changed to **ce[n]**.
5. In the Port field, enter the number of the port that you want the clock enable to apply to.
6. In the Formula field, enter the binary expression using the *bin()* function, where the Nth bit corresponds to whether this port should be enabled by the Nth trigger. For example, if you want port *n* output to be updated whenever either trigger 1 or trigger 3 occur, you would enter a value of *bin(101)* or 5 for the clock enable parameter.
7. Click **Apply** to update the parameter.
8. Click **OK** to accept the changes and dismiss the dialog box.

FDD Examples

This section offers the following examples that show how to use frequency-domain defined devices to define a variety of nonlinear circuit components. The examples include:

- [IQ Modulator](#)
- [Mixer](#)
- [Sample and Hold](#)

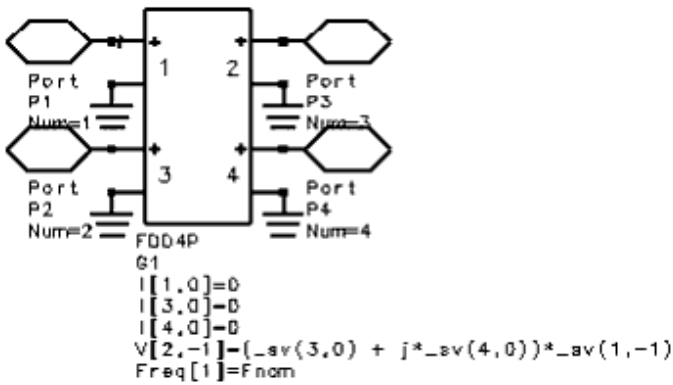
You can find these examples in the software under the Examples directory in this location:
Tutorials/FDD_Examples_wrk/networks

IQ Modulator

This example is under the Examples directory in the following location:

Tutorials/FDD_Examples_wrk/IQ_modulator

This is a simple, IQ modulator. The input signal is at port 1. The I and Q data (baseband, time-domain signals) are made available at ports 3 and 4, respectively.

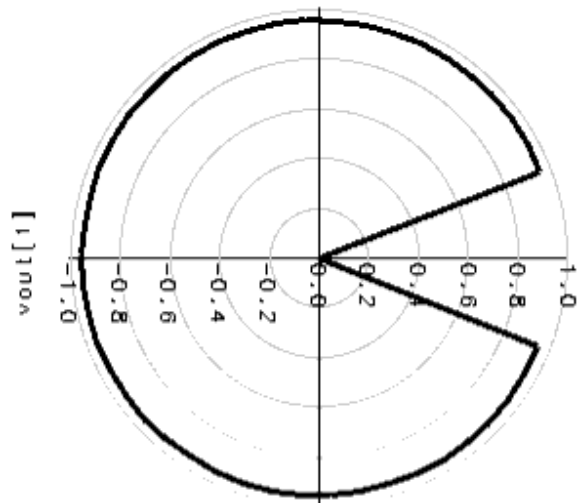


Note the following points:

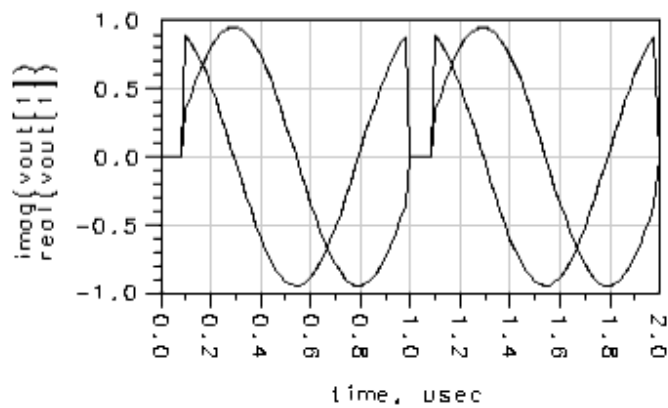
- I data is the baseband time-domain signal and is applied to port 3.
- Q data is the baseband time-domain data and is applied to port 4.
- The current equations for ports 1, 3, and 4 set the current at these ports to zero. These ports are treated as open circuits at all frequencies.
- The voltage equation at port 2 equates the spectral voltage at port 2 at frequency $Freq[1]$ to the baseband spectral voltage at port 3 (the I signal) plus j times the baseband spectral voltage at port 4 (the Q signal) multiplied by the spectral voltage at port 1 at frequency $Freq[1]$.
- Note the use of -1 in the left side of the voltage equation and in the function $_{sv}(1, -1)$. The minus sign is required when referring to the index of a carrier defined using the $Freq$ parameter, in this case, the index 1 that identifies $Freq[1]$.
- $Freq[1]$ is a user defined parameter whose value is passed to the FDD.

The cell *IQmodTest* shows this device under test. I and Q modulation are applied to ports 3 and 4. A 1 V, 1 GHz signal is applied to the input. The modulated output is shown here.

Modulated output signal,
on polar plot.



time (0.000 sec to 2.000usec)

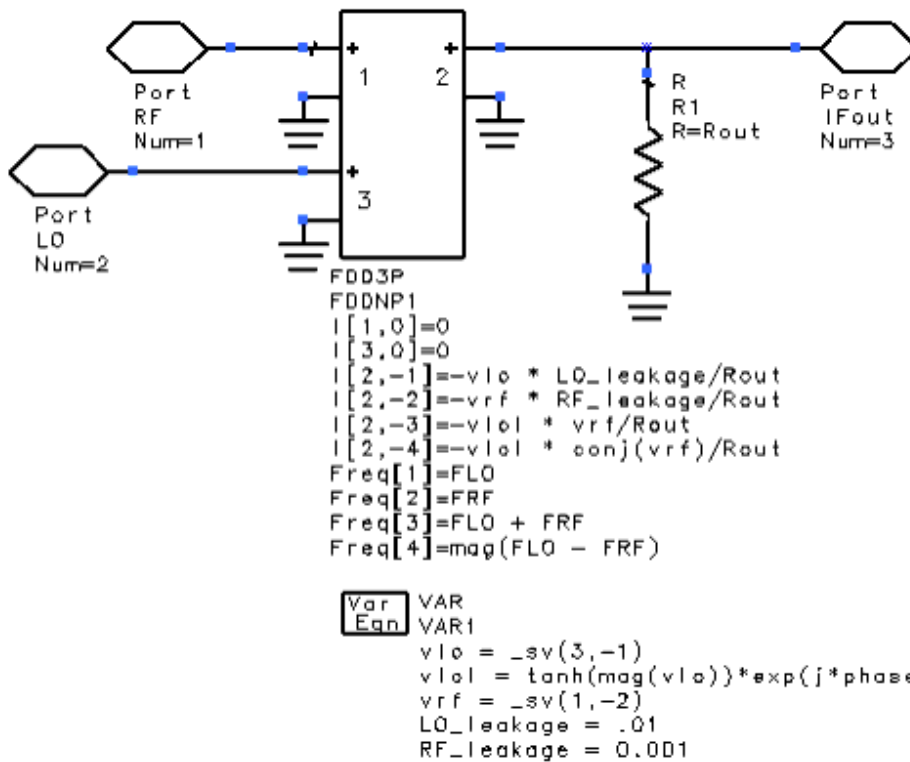


Mixer

This example is under the Examples directory in the following location:

Tutorials/FDD_Examples_wrk/FDDmixer

This is a simple, ideal mixer. It models upconversion, downconversion, LO leakage, RF leakage, and conversion gain compression with increasing LO amplitude

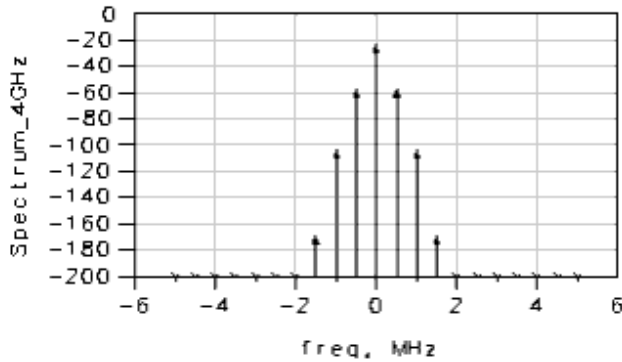


Note the following points:

- Like the IQ modulator example, ports 1 and 3 are open circuits for all frequencies.
- The signal at port 2 has four spectral components, which are defined with four current equations. The equations define:
 - LO leakage at frequency FLO
 - RF leakage at frequency FRF
 - The upconverted signal FLO+FRF
 - The downconverted signal, which is the magnitude of FLO - FRF
- Note the use of minus signs in the left side of the current equations. The minus sign is required when referring to the index of a carrier defined using the Freq parameter. In this case, -1, -2, -3, and -4 each refer to *Freq[1]*, *Freq[2]*, *Freq[3]*, and *Freq[4]*, respectively.
- The use of minus signs in the right side of the current equations is necessary because the equations define positive current flowing into each port of the FDD. Thus, the minus sign changes the direction of positive current.
- The variables *Rout*, *FLO*, and *FRF* are user-defined parameters whose values are passed to the FDD.

The cells *FDDmixerTest* and *FDDmixerTestEnv* show the mixer under test in a harmonic balance simulation and Circuit Envelope simulation, respectively. One result of the Circuit Envelope simulation is shown here.

Output spectrum centered at 4 GHz
This is the upconverted signal.



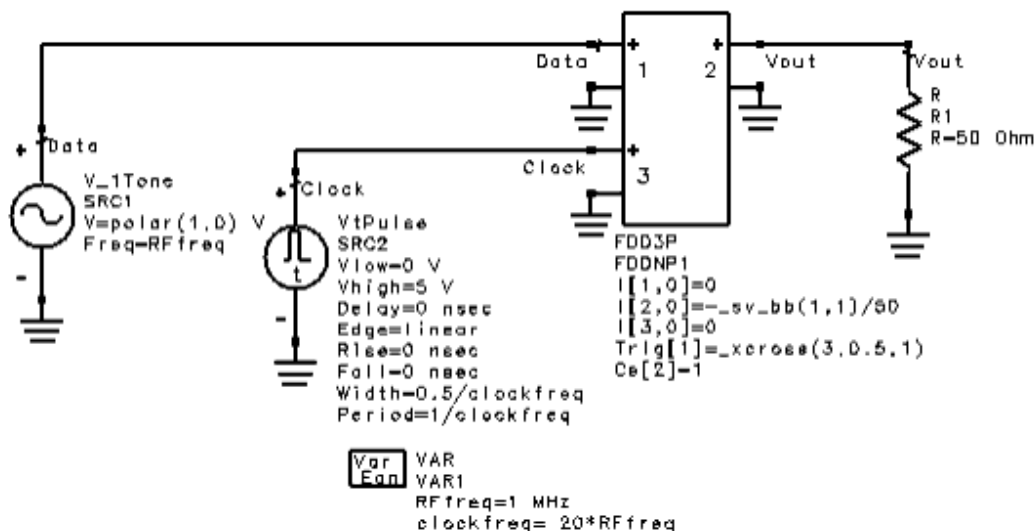
Sample and Hold

This example is under the Examples directory in the following location:

Tutorials/FDD_Examples_wrk/networks/SampleHold

This is a simple sample and hold device. The FDD samples the input data, in this case, a sine wave, once per rising edge of the clock, then holds the value so that the current is constant at the output. 20 samples are taken per period of the input signal.

This example also uses the trigger and clock enable features of the FDD.



Note the following points:

- Ports 1 and 3 are open circuits for all frequencies.
- The current equation for port 2 is based on the function `_sv_bb(port, trigger)`. This function, when passed a port number and trigger index, returns the total, real voltage at the port, at the last time the trigger occurred. So the current at port 2 is equal to the total, real voltage at port one at the last time trigger 1 occurred, divided by 50 ohms.

- The trigger parameter is based on the function `_xcross(port, threshold, direction)`. Given the values that are passed to this function here (3, 0.5, and 1), a trigger occurs if the baseband voltage at port 3 passes through 0.5 V, in the positive direction. For more information on triggers, refer to the section [Trigger Events](#).
- The clock enable parameter, *Ce*, enables the output of a port to change only when a specified trigger occurs. In this instance, it means that the output at port 2 will change only when trigger 1 occurs. This produces the "hold" effect of the sample and hold device. The trigger indices must be specified in binary format. For more information, refer to the section, [Output Clock Enables](#).

Building Signal Processing Models

ADS Ptolemy provides rich libraries of component models in Advanced Design System (ADS). However, you might want to create your own C++ component models to add to these libraries. ADS Ptolemy includes the Model Development Kit, a feature that allows you to create, compile, and link your models into ADS Ptolemy.

Once the shell of your model is built, the body of the model (the algorithm) will have to be written. To do this, you will need an understanding of the ADS Ptolemy Preprocessor Language, described in *Writing Component Models* (modbuild).

Note
Definitions for UC Berkeley Ptolemy terms, such as star and particle, are given in the *ADS Ptolemy Simulation* (ptolemy) documentation.

Advanced Model Building Functions

This section describes processes that provide you with the full features available in ADS Ptolemy models. This functionality, includes:

- Inheritance of model inputs, outputs, states, data, and methods from another star. These inherited properties are important for code reuse, increased code robustness, increased code quality, decreased code size, and reduced code testing requirements.
- Model states that use the enumerated type. These enumerated states are important for defining explicit state options at the design environment level.
- Model states that are hidden from the design environment. These hidden states are useful for local model parameter definitions.
- More detailed and flexible auto generation of AEL, bitmaps, and symbols required at the design environment level.
- Models that set the vendor field.

This section first walks you through developing a simple model and then provides more detail on certain topics.

Prerequisites to Model Development

Your UNIX or Windows system must have the appropriate C++ development software installed. For specific compiler requirements, see the system requirements table in the ADS installation documentation for the platform you are using:

- For UNIX, see *Before You Begin UNIX and Linux Installation* (install).
- For Windows, see *Before You Begin Windows Installation* (instalpc).

Creating a Simple Model Library

In UNIX Set the HPEESOF_DIR and PATH Environment Variables

1. Set the *HPEESOF_DIR* environment variable to wherever you've installed ADS.
2. Add the *\$HPEESOF_DIR/bin* directory to your PATH.

In Windows Use the Ptolemy Modelbuilder Shell

To activate the *Ptolemy Modelbuilder Shell* select: **Programs > Advanced Design System > ADS Tools > Ptolemy Modelbuilder Shell** from the Windows **Start** menu.

The advantage of using this window is that it automatically sets *\$HPEESOF_DIR* and adds *\$HPEESOF_DIR/bin* to the beginning of your path environment variable.

Alternatively, you can set the needed variables manually in the Command Prompt window using the *Set* command.

1. Set the *HPEESOF_DIR* environment variable to wherever you've installed ADS.
2. Add the *%HPEESOF_DIR%\bin* directory to your PATH.
3. Call VCVARS32.BAT (located in the VC\bin directory of your Microsoft Visual Studio installation).

Set Up the Modelbuilder Directory

A model build area can contain any number of libraries and star libraries. Each star library can contain many stars. Change directory to your home directory, and we'll create a model build area with a single star library for this tutorial:

```
hpeesofmb
```

The *hpeesofmb* command will create a directory called *adsptolemy* and copy some files there. Inside the *adsptolemy* directory are two files, *makefile* and *make-defs* which you shouldn't edit. Only one directory, *src*, exists at the beginning, which is where the source code for your libraries and stars will go.



Note

On Windows systems only, ADS mounts *\$HPEESOF_DIR\tools\bin* to */bin* as part of the *hpeesofmb* and *hpeesofmake* commands. If you have already mounted */bin* to another directory, you should be aware that ADS will unmount it. If you do not have a */bin* directory, a warning message may be displayed regarding the absence of the */bin* directory. This message can be ignored, however.

If you have a work area that was generated using the Ptolemy Modelbuilder prior to the ADS 2011 release, you must clean up and update your makefiles. To do this, do the following:

```
cd <dirname>
rm mk/*
cd ..
```

```
hpeesofmb -clean <dirname>
```

where *<dirname>* is the directory name where your model area was originally created using *hpeesofmb*, usually *adsptolemy*.

The src directory can be arbitrarily deep so that you can keep many star libraries and regular libraries in one model build area. In this example, we'll create a star library directly in src. A later section explains how to create more complicated src area.

Write a Model

We provide the code for many of our stars for you to look at in the directory *doc/sp_items* in the ADS installation. We'll copy the Sin star into the src area and edit it for our purposes:

In UNIX:

```
cd adsptolemy/src
cp $HPEESOF_DIR/doc/sp_items/SDFSin.pl SDFMySin.pl
vi SDFMySin.pl
```

and in Windows:

```
cd adsptolemy\src
copy %HPEESOF_DIR%\doc\sp_items\SDFSin.pl SDFMySin.pl
edit SDFMySin.pl
```

Change the name of the star. Find the line which says *name {Sin}*, and change it to *name {MySin}*. Change the code if you wish. For example, the star could compute $\sin() + 1$ by changing the go routine to:

```
output%0 << sin (double(input%0))+ 1;
```

Change the location of the star. Find the line which says *location {Numeric Math}* and change it to *location {My Stars}*.

Star files are named according to the convention *<Domain><Name>.pl*. The MySin star is in the SDF domain. The pl extension stands for Ptolemy Language.

Edit the make-defs

Every directory under src must contain a make-defs file. For this simple star, the default make-defs is almost correct. You need to find the line which says *PL_SRCS=* and change it to

```
PL_SRCS = SDFMySin.pl
```

Build the Shared Library

To build a shared library and install it into your build area, you need to run hpeesofmake.

**Note**

To build models for 64-bit platforms, you should use hpeesofmake64 instead of hpeesofmake. For more information, refer to [Building Models for 64-Bit Platforms](#).

Change directory back to the [Modelbuilder area](#) (your *adsptolemy* directory) and run this command:

```
hpeesofmake "debug=1"
```

**Note**

Due to changes in ADS 2011, if you have any shared libraries that were generated using the hpeesofmake command prior to the ADS 2011 release, you have to regenerate them using the ADS 2011 hpeesofmake command. To do this, refer to [Set Up the Area to Build Models](#), for details on how to run hpeesofmake - clean. If you fail to do this, your older shared libraries will *not* work in ADS 2011.

You must use hpeesofmake (which is actually GNU make) for all make commands---never make or nmake. The only exception is if you're developing in Windows with the Cygnus GNU-WIN32 tools. Refer to the later section [Platform-Specific Issues](#) for more details.

Building the shared library will take some time. If you do a listing of the adsptolemy directory, you'll see two new directories, lib.*arch* and obj.*arch*, where *arch* is an abbreviation for your architecture, e.g. *win32*.

To keep your src directory tidy, all compiled files are placed in an equivalent area in obj.*arch*. The libraries that will be needed by the simulator are placed in lib.*arch*. Since architecture dependent files are placed in different directories, you can do development for multiple architectures in one model build area.

The "debug=1" option above causes the library to be built as code that can be debugged. It is built as optimized code without it. You can also add the line `debug=1` to your make-defs to always build code that can be debugged.

**Note**

On Windows, do not use "debug=1" with hpeesofmake to share user compiled models with other users that do not have Microsoft Visual Studio installed.


Create the Library

To use your star in the Signal Processing schematic, you must generate the associated AEL, bitmap, and symbol and create the library that can be loaded into ADS. Run this command from your [Modelbuilder area](#) (your *adsptolemy* directory):

```
hpeesofmake oalib
```




ADS will briefly appear to create the library. Please do not close any ADS windows - they will disappear automatically after ADS is done.

 On UNIX you must set DISPLAY as if you were running ADS for this step to work.

The AEL code describes your model to the ADS design environment. You must regenerate AEL whenever you change an exterior aspect of your model, for example, its name, ports, parameters, or location.

The bitmap is the picture of your model which appears in the palette on the left side of the design environment. The symbol is the picture which actually gets placed in the schematic.

The bitmaps and symbols created by the make system are simple but you can edit them further. They will be the right size and have the appropriate number and type of pins. Symbols can be edited in ADS. Bitmaps can be edited with the bitmap program in UNIX or the Paint program in Windows.

 The build system will not overwrite existing symbols or bitmaps; it will only create symbols and bitmaps for the stars which you have added since the last time you ran it. If you want to force the creation of a particular bitmap or symbol, manually remove the appropriate file.

Simulate Your Model

Before starting ADS, set the ADSPTOLEMY_MODEL_PATH environment variable to point to your model build area. The simulator uses this variable to find your libraries and bitmaps. The variable is a colon delimited path in UNIX and a semicolon delimited path in Windows. For this example, set it to \$HOME/adsptolemy in UNIX or c:\users\default\adsptolemy in Windows.

Now start ADS. You'll need to add the component library created under oalibs directory in the model build area to your workspace and then you'll see your star on the palette and in the component browser under wherever you set the location field above.

You can rebuild your shared library while ADS is running if you first choose *Simulate > Stop and Release Simulator*.

Sharing Your Stars

Other users can simulate designs with your models by adding your directory to their ADSPTOLEMY_MODEL_PATH. If you are not on the same network, you can send them the entire contents of your model build area, minus the src and mk directories if you wish to protect those directories. You can also send them the .pl files and the make-defs and ask them to recompile your models.

The src Directory and make-defs in More Detail

If you are trying for the first time to create a user-defined model for ADS Ptolemy Simulation in ADS, please review the introductory information above beginning with [Building Signal Processing Models](#).

In a customized .pl file, there should be a statement that defines which library/palette group the user-defined model should go to. The statement `location{MyStars}` is just for this purpose. By defining `location{MyStars}`, it means that the user-compiled model will be found in the palette group called MyStars, and it will also be found in the library category MyStars in the library browser. You can also use the statement as `{Numeric,MyStars}`. By doing this the palette group will show up as Numeric MyStars. In the library browser, the MyStars category will become a subgroup below the group Numeric.

It is acceptable to have many .pl files in the src directory. The file structure in the adsptolemy directory in your home directory has nothing to do with which library/palette group the user-defined model resides in.

It is possible to create sub-folders below the adsptolemy/src directory. You can create a hierarchy of locations for different sets of models. For example, you can create a structure such as,

directory	subdirectories
adsptolemy/src	/my_lib1
	/my_lib2
	/my_lib3
	/my_lib4

Then you can place different sets of .pl files into the various subdirectories my_lib1 through my_lib4. To compile these models, each my_libx directory should have a file called make-defs. You can copy the make-defs from the adsptolemy/src level into the sub-directories my_lib1, my_lib2, etc. When these .pl files are in place, there are four things you must do to prepare them for compilation:

- Each .pl file, defines a palette/library group for the user-defined model using the `location{}` statement.

In the make-defs file in the src level, make sure that the variable `PL_SRCS` is equal to blank spaces.

- In the make-defs file in the src level, make sure to add the variable

```
DIRS = my_lib1 my_lib2 my_lib3 my_lib4
```

This defines that all the .pl files in the sub-directories my_libx will be compiled.

- In the make-defs file in each sub-directory such as my_lib1, list the .pl files within that directory in the variable `PL_SRCS`. For example,

```
PL_SRCS = model1 model2 model3
```

- In the make-defs file in each sub-directory, such as my_lib1, the variable `STAR_MK` must be assigned a unique library name. For example in the make-defs file in the sub-directory my_lib1, you can have `STAR_MK = usermodels_my_lib1`. In the make-defs file in the sub-directory my_lib2, one can have

STAR_MK = usermodels_my_lib2, etc.

Once you have done the above, use the procedure in [Creating a Simple Model Library](#) above to compile all the models in the sub-directories.



Note

When a hierarchical structure is defined such as one defined above, the any .pl files in the src level will not be compiled. Even if the make-defs file in the src level have defined these .pl files in the PL_SRCS variable.

The model creation procedure above includes a comment about star libraries and conventional libraries. The difference between them is as follows:

- **star library** refers to a sub-directory below the src level, which contains .pl files. A star basically means a model that can be placed in an ADS design window. A .pl file is a source file that describes the model which is written in a C-like language called Ptolemy language.
- **conventional library** refers to a subdirectory below the src level, which contains .c files. These are standard C source which can be referenced by other .pl files in another sub-directory under src so that algorithms written originally in C can be used directly in a user-defined Ptolemy model.

Star library name, conventional library name, names of sub-directories under the src level, and library names assigned to STAR_MK variables in make-defs files in sub-directories under src do not determine where a model will be located in the palette/library browser. The location{} statement in the .pl defines the name of the palette/library group in which the user-defined model can be found.

Variables

As mentioned above, the src directory can have arbitrary depth. The build system will recurse over your entire tree. Each directory must have a make-defs file; the makefile is built automatically from this file.

Directories containing other directories should define the DIRS variable in their make-defs to a space-separated list of the directories to recurse into. For example, if your src directory contains two directories, foo and bar, the contents of the make-defs in the src directory would be:

```
DIRS = foo bar
```

Two kinds of libraries can be built by the build system: star libraries and conventional libraries. To build a star library, set the PL_SRCS variable to a space-separated list of your .pl files and the STAR_MK variable to the name of the star library. For example, part of a make-defs that builds a star library with two models might be:

```
STAR_MK = myfilter
PL_SRCS = filter1.pl filter2.pl
```

To build a conventional library, set the SRCS variable to a space-separated list of your .c, .cc, and .cxx files and the PTLIB variable to the name of the library. The STAR_MK and PTLIB variables are mutually exclusive.

You can set other make-defs variables in order to control compilation and linking. Append compilation flags to the variables CPPFLAGS, CFLAGS, and CXXFLAGS to affect preprocessing, C compilation, and C++ compilation. Since all stars are written in C++, use the CXXFLAGS to control star flags. For example,

```
CPPFLAGS += -DFAST
CFLAGS += -O4
CXXFLAGS += -O4
```

Add to the include path by adding directories to the variable INCLUDEPATH, a space-separated list. Additional objects can be linked into your library by appending to the OBJS variable. Additional sources can be compiled and linked in by appending to the SRCS variable. For example,

```
INCLUDEPATH += /libtree/headers
SRCS += myutilities.c moreutilities.cxx
OBJS += /libtree/objs/tree$(OBJ_SUFFIX)
```

1. OBJ_SUFFIX will expand to .o on Unix and .obj on Windows
Linking is manipulated with the variables LIBSPATH, LIBS, and LIBSOPTION. Similar to INCLUDEPATH, LIBSPATH is a space-separated list of directories where the linker should look for libraries. LIBS is a list of the libraries themselves. LIBSOPTION allows any arbitrary flags to be added to the link. For example,
LIBSPATH += /libtree/libs
LIBS += tree m
2. The tree library and the math library are linked in by the above.
Because the make system will automatically set appropriate values for most of the variables, you should almost always append to the variable with += rather than setting it with =.

Dependencies

You must tell the make system on which ADS libraries your library depends. This will cause the make system to add the appropriate directories to your include path. In addition, the library will be built in such a way that the dependent libraries are loaded along with your library.

To use any of the ADS headers in adsptolemy/src, you must set a particular make-defs variable to 1. Each directory has a corresponding variable according to the table:

Dependencies

Directory	make-defs Variable
adsptolemy-kernel/compat	(always included)
adsptolemy-kernel/kernel	KERNEL
numeric/kernel	SDFKERNEL
numeric/base/stars	SDFSTARS
numeric/dsp/stars	SDFDSP
numeric/libptdsp	PTDSP
timed/kernel	TSDFKERNEL
timed/base/stars	TSDFSTARS
matrix/base/stars	SDFMATRIX
fixpt-analysis/base/stars	SDDFIX
controls-displays/tcltk/ptklib	PTK
controls-displays/tcltk/stars	SDFTK
instruments/stars	SDFINSTKERNEL

At a minimum, you will need to set the variable corresponding to the do main for which you are building stars: *SDFKERNEL* for SDF and *TSDFKERNEL* for TSDF. Dependencies are transitive so if you depend on A, and A depends on B, the make system will require you to depend on both A and B.

When the *SDFINSTKERNEL* is set in the make-defs file (*SDFINSTKERNEL*=1), it will include the *SDFInstrument.h* header file with the other dependencies at compilation.

ADS provides the code for many ADS Ptolemy stars in the *doc/sp_items* directory. These are the stars for which you can click the C++ Code button in the *Signal Processing Components* documentation. To derive a star of your own from one of these, find the header of the star in one of the above locations, and set the appropriate variable.

Debugging Your Model

Debugging a program that loads dynamic libraries at run time is a more difficult task than debugging a conventional program. The symbol tables for the dynamic libraries must be manually loaded before the debugger can set break points in those libraries. You must have compiled your code with the debug flag on as explained earlier.

Before you start debugging, you'll need to be able to run simulations from the command line, outside of ADS.

Running Simulations from the Command Line

Each time you simulate from ADS, a file called *netlist.log* is created in your workspace directory. This file completely describes your schematic and can be passed to the simulator on the command line. Note that the format of this file is not guaranteed to remain the same in future versions of ADS.

Before you can execute the simulator, certain environment variables must be set so that

the simulator can find all its shared libraries. When you ran hpeesofmb, two scripts called mbsetvars and mbsetvars64 (mbsetvars.bat and mbsetvars64.bat on Windows) were created in the bin directory.

Under UNIX, you will have to evaluate their output to set the appropriate variables. Assuming you're building models in your home directory, type:

```
cd ~/adsptolemy/bin
eval `./mbsetvars -u` (if you are building 32-bit libraries and are using
sh/bash/ksh shell)
eval `./mbsetvars -c` (if you are building 32-bit libraries and are using csh/tcsh
shell)
eval `./mbsetvars64 -u` (if you are building 64-bit libraries and are using
sh/bash/ksh shell)
eval `./mbsetvars64 -c` (if you are building 64-bit libraries and are using
csh/tcsh shell)
```

Under Windows, the scripts are .bat files so you can run them directly:

```
cd c:\users\default\adsptolemy\bin
mbsetvars (if you are building 32-bit libraries)
mbsetvar64 (if you are building 64-bit libraries)
```

The mbsetvars scripts have the path to your model build area encoded in them, so you should not use these scripts from a different model build area. If you move the model build area, regenerate the scripts by removing them and then running the hpeesofmb command again.

Now you should be able to run your simulation from the command line by moving to your workspace's data subdirectory and running:

```
hpeesofsim ../netlist.log
```

Debugging Under Windows

After running the *mbsetvars* script to set the needed variables, start the Visual C++ (Visual Studio 2008) Debugger from the command line with the *devenv* command. If you start it from the Start menu, it will not work because it will not inherit the environment variables set by mbsetvars.

1. From the File menu, choose *Open Solution*.
2. In the *Open Solution* dialog box, click the *Files of type* list box and choose *Executable Files*.
3. Locate the executable (hpeesofsim.exe) from the bin directory of the ADS installation and click **OK**. This creates a temporary solution that contains the executable. Before using an execution command (such as *Start*), you must save the solution.
4. From the *Project* menu, select *hpeesofsim Property Pages*. On the property page, set the Working Directory to your ADS workspace/data directory and set the Command Arguments to *..\netlist.log*.
5. Set breakpoint and debug the newly created model.

Debugging Under UNIX

After evaluating the output of the `mbsetvars` script to set the needed variables, start the debugger on the `bin/hpeesofsim` binary. On Solaris, use the regular debugger, `dbx`. On Linux, use the GNU `gdb`.

The GNU `gdb` supports deferred breakpoints setting for future shared libraries loading. This will let users to set breakpoints after the debugger has started but before loading any user shared libraries. Run the simulator in your workspace's data directory with an argument of `../netlist.log`. The debugger will stop when any breakpoint is reached.

With `dbx`, the debugger needs to know that a program will use a library when it runs in order to set breakpoints. Simply run the simulator with an argument of `../netlist.log` once and the debugger will capture the list of loaded libraries. The libraries will remain loaded after the simulator process ends and you can set breakpoints in them before rerunning the program for debugging.

STL

Advanced Design System uses and ships STLport (<http://www.stlport.org/>) to support ANSI C++ Standard Library on multiple platforms. Models built using Ptolemy Modelbuilder will always be linked with STLport. The following sections describe some rules regarding the use of STLport.

Namespace std

The `std` namespace is supported by STLport, which means anything defined in `std` namespace will be using STLport implementation rather than the native compiler implementation. That includes `iostream`, `string`, `list`, `vector`, etc. Due to a problem on HP-UX compiler, `iostream` may output incorrect data. The work around is to explicitly put

```
using namespace std;
```

in the source code AND use `std::cerr`, `std::cout` with `std::` prefix.

To avoid conflict with old native compiler implementation of `iostream`, etc., always use the correct non `.h` `#include` header files, that is, `<iostream>` instead of `<iostream.h>`.

In the case of a ptolemy `pl` file, the following example code shows how to set the namespace `std`.

```
defstar {
    name { testModel }
    .
    .
    code { using namespace std; }
    .
    .
}
```


}

This way, only the generated .cc file will have namespace std set. It is not a good practice to have namespace defined in the .h file because derived class including the .h should define the use of namespace on their own. If the class definition requires the use of std, such as data member using STL list, etc., you can use the following example code to set the namespace in the .h file.

```
defstar {
    name { testModel }
    .
    header { using namespace std; }
    .
}
```

Linking with Third Party Library

If the third party library is built with STLport, there should be no problem linking in with models built using Ptolemy Modelbuilder. A problem exists when the third party library is built without using STLport.

- Passing of std:: data types across libraries with different STL implementations is not allowed.
- Classes with std:: data members in native STL libraries cannot be linked in and used in Advanced Design System, for example, ThirdParty.h

```
#include <list>
class myClass {
    myclass();
    ~myclass();
    void push(int a);
    int pop();
protected:
    list<int> i;
};
```

If *myClass* is built without using STLport, it will not work when a model built using Ptolemy Modelbuilder is referencing *myClass* directly. It is due to the use of *list* in std::. The workaround is to write a wrapper class to wrap around myClass so that it does not define any STL, or std:: constructs in the header file and build this new wrapper class without using STLport. As an example of ThirdParty.h, a ThirdPartyWrapper.h will appear as follows:

```
class myClass; //forward declaration
class myClassWrapper {
    myClassWrapper();
    ~myClassWrapper();
    void push(int a);
    int pop();
protected:
    myClassWrapper* myC;
};
```

A Model using Ptolemy Modelbuilder can now link with this wrapper library to use the

class. For example:

```
defstar {
    name { myModel }
    domain { SDF }
    hinclude { "ThirdPartyWrapper.h" }
    protected {
        MyClassWrapper mcw;
    }
    constructor {
        mcw.push(0);
    }
    .
    .
}
```

Platform-Specific Issues

Building Models for 64-Bit Platforms

The 64-bit platforms that ADS Ptolemy supports are Windows, Linux and Solaris. To build on these platforms, you should use `hpeesofmake64` instead of `hpeesofmake`.

On Solaris, if you want to support both 32-bit and 64-bit simulations, you should also run `hpeesofmake install` to build the required 32-bit libraries.

On 64-bit Linux platforms, to build 32-bit libraries use the command:
`hpeesofmake "CXX=g++ -m32"`

Windows

Use the *Ptolemy Modelbuilder Shell* to build 32-bit libraries and the *Ptolemy Modelbuilder Shell (64 bit)* to build 64-bit libraries.

Make sure that you are pointing to the 32-bit compiler to build 32-bit libraries and to the 64-bit compiler to build 64-bit libraries.

The build system under Windows uses the Cygnus GNU-Win32 tools internally. You may build with a normal MS-DOS shell, the MKS toolkit, or the Cygnus GNU-Win32 tools. If you use either of the first two, you should use `hpeesofmake` as the documentation describes. But if (and only if) you have the Cygnus tools installed and are building under the Cygnus bash shell, you should use the *make* command to build.

Windows Vista and Windows 7

On Windows Vista and Windows 7, you must have Administrative rights with *User Account Control (UAC)* disabled.

Consult Windows Vista or Windows 7 help to disable UAC.

UNIX

On UNIX you must set DISPLAY as if you were running ADS in order to create the ADS2011 library with your components, since that step requires running ADS.

Operating System Defined Types

In order to separate the use of INT, FLOAT, etc. and the OS defined types, AgilentPtolemy namespace is defined. If you are using a named Ptolemy type in your code, you will need to use it in an AgilentPtolemy namespace, such as AgilentPtolemy::DataType, AgilentPtolemy::INT, AgilentPtolemy::FLOAT, etc. For an example of how the AgilentPtolemy namespace is used, refer to the *SDFSsimpleNumericSink* (modbuild) example.

Writing Component Models

As described in *Building Signal Processing Models* (modbuild), you can build your own component models to supplement the large libraries included with ADS Ptolemy. This section describes how to write the body of these models, and includes:

- [Using the ADS Ptolemy Preprocessor Language](#)
- [Writing C++ Code for Stars](#)
- [Writing Timed Components](#)
- [Writing Sink Models](#)

Using the ADS Ptolemy Preprocessor Language

Since the stars in ADS Ptolemy were designed to be as generic as possible, many complicated functions can be realized by a galaxy. Even so, no star library can possibly be complete. You may have to design your own stars. The ADS Ptolemy preprocessor language makes this easier.

The ADS Ptolemy preprocessor was created to make it easier to write and document star class definitions to run under ADS Ptolemy. Instead of writing all the class definitions and initialization code required for an ADS Ptolemy star, the user can concentrate on writing the action code for a star and let the preprocessor generate the standard initialization code for portholes, states, etc. The preprocessor generates standard C++ code, divided into three files:

- A header file with a *.h* extension.
- An implementation file with a *.cc* extension.
- An xml file with a *.pl.xml* extension for auto-documentation generation.

Rectangular Pulse Star Example

To make things clear, let's start with an example: a rectangular pulse star in the file *SDFRect.pl*. See [SDFRect.pl File](#). This is the code for an actual star. The code for more examples can be found in `$HPEESOF_DIR/doc/sp_items` for UNIX systems or `%HPEESOF_DIR/doc/sp_items` for PC platforms.

From the file *SDFRect.pl*, the model building process creates the files *SDFRect.h*, *SDFRect.cc*, and *SDFRect.pl.xml*. The names are determined by concatenating the domain and name fields. These files define a class named *SDFRect*. The example code is as follows:

```
defstar {
    name { Rect }
    domain { SDF }
    desc { Rectangular pulse output }
    explanation {
```

User-Defined Models

Generate a rectangular pulse of height "height" (default 1.0). and width "width" (default 8). If "period" is greater than zero, then the pulse is repeated with the given period.

```
}
version { @(#) $ $Revision: 1.18 $ $Date: 2001/03/23 22:19:18 $ }
ucb-version { @(#)SDFRect.pl      2.10 6/25/96}
author {your_name}
copyright {
Copyright (c) Agilent Technologies 2001
Copyright (c) 1990-1995 The Regents of the University of California.
All rights reserved.
See the file $ROOT/ucb-copyright for copyright notice,
limitation of liability, and disclaimer of warranty provisions.
}
vendor { AgilentEEsof }
location { Numeric, Sources }
output {
    name { output }
    type { float }
}
defstate {
    name { Height }
    type { float }
    default { 1.0 }
    desc { height of rectangular pulse }
}
defstate {
    name { Width }
    type { int }
    default { 8 }
    desc { width of rectangular pulse }
}
defstate {
    name { Period }
    type { int }
    default { 0 }
    desc { if greater than zero, repetition period of pulse stream}
}
defstate {
    name { Count }
    type { int }
    default { 0 }
    desc { Internal counting state. }
    attributes { A_NONSETTABLE|A_NONCONSTANT }
}
setup {
    Count = 0;
}
go {
    double t = 0.0;
    if (int(Count) < int(Width)) t = Height;
    output%0 << t;
    Count = int(Count) + 1;
    if (int(Period) > 0 && int(Count) >= int(Period)) Count = 0;
}
}
```

SDFRect.pl File

Only one type of declaration may appear at the top level of an ADS Ptolemy language file: a *defstar*, used to define a star. The defstar section is itself composed of subitems that define various attributes of the star. All subitems are of the form:

keyword {body}

where the body may itself be composed of sub-subitems, or may be C++ code (in which case the ADS Ptolemy language preprocessor checks it only for balanced curly braces).

Note that the keywords are not reserved words. They may also be used as identifiers in the body.

Items Defining a defstar

The following table provides an alphabetical list of the items that can appear in a defstar directive, including a summary of directives.

Summary of Items Used to Define a Star

Keyword	Summary	Required	More Information
acknowledge	The names of other contributors to the star	No	acknowledge
attributes	Attributes for the star	No	attributes (for Stars)
attributes	Attributes for PortHoles	No	attributes (for PortHoles)
attributes	Attributes for the States	No	attributes (for States)
author	The name(s) of the author(s)	No	author
begin	C++ code to execute at start time, <i>after</i> __ the schedule setup	No	begin
ccinclude	Specify other files to include in the .cc file	No	ccinclude
code	C++ code to include in the .cc file outside the class definition	No	code
conscalls	Define constructor calls for members of the star class	No	conscalls
constructor	C++ code to include in the constructor for the star	No	constructor
copyright	Copyright information to include in the generated code	No	copyright
derived	Alternate form of <i>derivedfrom</i>	No	derived
derivedfrom	The base class, which may also be a star	No	derivedfrom
desc	Alternate form of <i>descriptor</i>	No	desc
descriptor	A short summary of the functionality of star	No	descriptor
destructor	C++ code to include in the destructor for the star	No	destructor
domain	The domain and the prefix of the name of a class	Yes	domain
explanation	Full documentation optionally using <i>troff</i> , <i>_eqn_</i> , and <i>_tbl_</i> formats	No	explanation
footnotes	Optional field to include some footnotes	No	footnotes
go	C++ code to execute when the star fires	No	go
header	C++ code to include in the .h file, before the class definition	No	header
hinclude	Specify other files to include in the .h file	No	hinclude
htmldoc	Full documentation optionally using <i>troff</i> , <i>_eqn_</i> , and <i>_tbl_</i> formats	No	htmldoc
inmulti	Define a set of inputs	No	inmulti
input	Define an input to the star	No	input
location	Component library (palette) name where user will find the star	No	location

User-Defined Models

method	Define a member function for the star class	No	method
name	The name of the star and the root of the name of the class	Yes	name
outmulti	Define a set of outputs	No	outmulti
output	Define an output from the star	No	output
private	Define private data members of the star class	No	private
protected	Defined protected data members of the star class	No	protected
public	Define public data members of the star class	No	public
range	Define range of the parameter	No	range
setup	C++ code to execute at start time, before the scheduler setup	No	setup
state	Define a state or parameter	No	state
symbol	Associate symbol to a parameter for better documentation	No	symbol
version	Version number and date	No	version
wrapup	C++ code to invoke at the end of a run (if no error occurred)	No	wrapup
vendor	Name of company that authors component. All shipped with ADS Ptolemy are marked Agilent EEsof	No	vendor
Indicates a minimum set of the most useful items.			

An alternate form for the state directive is defstate. The subitems of the *state* directive are summarized in the following table, together with subitems of other directives.

Directive Subitem

Items	Subitems and Descriptions	Required	Page
inmulti, input	<i>name</i> (Name of port or group of ports) <i>type</i> (Data type of input and output particles) <i>descriptor</i> (Summary of function of the input) <i>numtokens</i> (Number of tokens consumed by the port; useful only for dataflow domains)	Yes No No No	inmulti, input
method, virtual method, inline method, pure virtual method, inline virtual method	<i>name</i> (Name of the method) <i>access</i> (Private, protected, or public) <i>arglist</i> (Arguments to the method) <i>type</i> (Return type of the method) <i>code</i> (C++ code defining the method)	Yes No No No If not pure	method
outmulti, output	<i>name</i> (Name of port or group of ports) <i>type</i> (Data type of output particles) <i>descriptor</i> (Summary of the functions of output) <i>numtokens</i> (Number of tokens produced by port; useful only for dataflow designs)	Yes No No No	outmulti, output
state	<i>name</i> (Name of the state variable) <i>type</i> (Data type of the state variable) <i>default</i> (Default initial value; always a string) <i>descriptor</i> (Summary of function of state) <i>attributes</i> (State attributes for simulator) <i>units</i> (Type of dimensional units associated with state) <i>enumlist</i> (list of enumeration options) <i>enumlabels</i> (list of alternate names for enumeration options) <i>extensions</i> (list of file extensions for a filename state - default is txt)	Yes Yes No No No No No No No No	state

In the text that follows, items are listed in the order in which they typically appear in a star definition (although they can appear in any order). In this list, syntax and descriptive notes are also included.

name

Required item. Syntax:

name {identifier}

Together with the domain, this item provides the name of the class to be defined and the names of the output files. Case is important in the identifier.

domain

Required item, specifying the domain, such as SDF or TSDF. Syntax:

domain {identifier}

where identifier specifies the domain (again, case is important).

derivedfrom

This optional item indicates that the star is derived from another star. Syntax:

derivedfrom {identifier}

where identifier specifies the base star. The *.h* file for the base class is automatically included in the output *.h* file, assuming it can be located (you may need to add -I options to the makefile).

For example, the LMS star in the SDF domain is derived from the FIR star. The full name of the base class is SDDFIR, but the derivedfrom statement allows you to say either

derivedfrom {FIR}

or

derivedfrom {SDDFIR}

The derivedfrom statement may also be written derivedFrom or derived. Note that it is not possible to derive stars across domains.

descriptor

This optional item defines a short description of the class. This description is displayed by the Advanced Design System design environment for this star in the Library list. It has the syntax

descriptor {text}

where text is simply a section of text that will become the short descriptor of the star. You can also write desc instead of descriptor. A principal use of the short descriptor is to get on-screen help. The following are legal descriptors:

desc {A one line descriptor.}

or

desc {A multi-line descriptor. The same line breaks and spacing will be used when the descriptor is displayed on the screen.}

By convention, in these descriptors, references to the names of states, inputs, and outputs should be enclosed in quotation marks. If the descriptor seems to get long,

augment it with the explanation or `htmldoc` directive, explained below. However, it should be long enough so that it is sufficient to explain the function of the star.

version

This optional item contains entries as follows.

```
version {@(#) $Source: <dir>/my_model.pl $ $Revision: number $ $Date:
YR/MO/DA $}
```

where the `<dir>` is the source code control directory, the number is a version number, and the YR/MO/DA is the version date.

author

This optional entry identifies the author or authors of the star. Syntax:

```
author {author1, author2 and author3}
```

Any set of characters between the braces will be interpreted as a list of author names.

acknowledge

This optional entry attaches an acknowledgment section to the documentation. Syntax:

```
acknowledge {arbitrary single line of text}
```

attributes (for Stars)

This optional entry defines star attributes with syntax:

```
attributes {attribute | attribute | ...}
```

where attributes are separated by the symbol `|`.
The only possible attribute is:

S_HIDDEN The star is invisible in the design environment. This is typically used for stars that are used only as base stars for other stars.

By default, a star is visible.

attributes (for PortHoles)

This optional entry defines PortHole attributes with syntax:

```
attributes {attribute | attribute | ...}
```

where attributes are separated by the symbol |.

Possible attributes are:

P_HIDDEN The port is invisible in the design environment.

P_VISIBLE The port is visible in the design environment.

P_OPTIONAL The port can be left unconnected. Note, the star code should not read data if the port is not connected. To test whether a port is connected or not, call the function `Port_Name.far()`, where `Port_Name` is the name of the port. If the function returns a NULL pointer, the port is not connected, otherwise it is connected.

P_REQUIRED The port is required to be connected.

By default, all PortHoles are visible and require a connection. Note that the simulator will connect BlackHole models to any unconnected output PortHoles. The BlackHole model itself is hidden.

attributes (for States)

This optional entry defines state attributes with syntax:

```
attributes {attribute | attribute | ...}
```

where attributes are separated by the symbol |.

Possible attributes are:

A_CONSTANT The state is constant during execution of the star.

A_NONCONSTANT The state can change during the execution of the star.

A_SETTABLE The state is visible in the design environment.

A_NONSETTABLE The state is invisible in the design environment.

A_SWEEPABLE The state can be optimized or swept.

A_NONSWEEPABLE The state can neither be optimized nor swept.

A_SCHEMDISPLAY The state is visible and editable on the ADS schematic.

A_NOSCHEMDISPLAY The state is visible and editable only in the Edit Component Parameter dialog box.

By default a State is constant, settable, sweepable, and displayed on the schematic.

copyright

This optional entry attaches a copyright notice to the *.h*, *.cc*, and *.t* files. Syntax:

```
copyright {copyright information}
```

For example, we use the following:

```
copyright {Copyright (c) Agilent Technologies 2000. All rights reserved.}
```

The copyright can span multiple lines, just like a descriptor.

location

This optional item is the name of the schematic library in which the user will find the star. Syntax:

```
location {<main libraryname,>libraryname}
```

where *libraryname* is the location of the star in the Advanced Design System design environment Signal Processing schematic and under Help > Topic and Index > Components> Signal Processing. The optional *main libraryname* is a super location followed by a comma. For example:

```
location {Signal Processing Library}
location {Numeric, Sources}
```

No more than two levels is allowed in the hierarchy.

explanation

This optional item is used to give a longer explanation of the star's function. Syntax:

```
explanation {
body
}
```

footnotes

This field is an optional field to include some footnotes in the pl file. There should be only one footnotes section per pl file. For Example.

```
defstar {
  name { _Clock }
  domain { TSDF }
  derivedfrom { basesource }
  footnotes {
    dagger.gif This parameter must satisfy the condition Tstep
    lt_equal.gif
    min (DutyCycle * Period, (1-DutyCycle) * Period)
```

```

}
.
defstate {
name { DutyCycle }
range { (0,1) dagger.gif }
type { float }
default { 0.5 }
desc { clock duty cycle }
}
.
}

```

range

This is an optional field for each state parameter. Developers can utilize this field to specify the range of a parameter. The following is how a developer specifies the range of the "Delay" parameter in the component TSDF_DelayRF.pl file.

```

defstar {
    name { _DelayRF }
    desc { Time delay }
    .
    .
    defstate {
    name { Delay }
    *range* { {-1} or [Tstep,inf) }
    type { float }
    .
    .
    }
}

```

The above range statement means the Delay can be -1 or any number between Tstep and positive infinity, including Tstep.

The following is the parsing rule for Range Syntax, the bold face means actual reserved keywords. The rule was originally from Verilog-A syntax and has been modified to include more complicated usage, such as set syntax.

```

ptolemy_range ::=
    ptolemy_range or value_range_specifier
value_range_specifier ::=
    start_paren expression1 , expression2 end_paren
    | { comma_separate_list }
comma_separate_list ::=
    comma_separate_list , const_expression
    | const_expression
start_paren ::=
    [
    | (
end_paren ::=
    ]
    | )
expression1 ::=
    constant_expression | -inf
expression2 ::=
    constant_expression | inf

```

Examples

- {-1} or (0, inf)
- {1, 2, 4, 8, 16}
- (-inf, inf)
- [LE+PL+TE, inf)

symbol

This field is an optional field for each state parameter to aid the documentation process. A developer can use this field to give a symbol to a particular parameter state. That symbol can later be used in equations to better explain the functionality of a component in the documentation.

Example

```
defstate {
    name { Phase }
    symbol { theta.gif }
    range{ (-inf,inf) }
    type { float }
    default { 0 }
    desc { reference phase in degrees }
    units { ANGLE_UNIT }
}
```

htmldoc

This optional item is used to give a longer explanation of the star's function. Syntax:

```
explanation {
body
}
```

state

This optional item is used to define a state or parameter. The following is an example of a state definition:

```
state {
    name {gain}
    type {int}
    default {1.0}
    units {UNITLESS_UNIT}
    desc {output gain}
    attributes {A_CONSTANT | A_SETTABLE}
}
```

The following ten types of subitems may appear in a state definition, in any order: *name*, *type*, *default*, *desc*, *units*, *enum*, *enumlist*, *enumlabel*, *extensions*, *attributes*.

- The *name* field (required) is the name of the state.
- The *type* field (required) is its type, which may be one of *int*, *float*, *string*, *complex*, *fix*, *intarray*, *floatarray*, *complexarray*, *precision*, *stringarray*, *filename*, *enum*, *query*, or *boolean*. Case is ignored for the type argument.
- The *default* field (optional) specifies the default initial value of the state. Its argument is either a string (enclosed in quotation marks) or a numeric value. The preceding entry could equivalently have been written:

```
default { "1.0" }
```

Furthermore, if a particularly long default is required, as for example when initializing an array, the string can be broken into a sequence of strings. The following example shows the default for a *ComplexArray*.

```
default {
"(-.040609,0.0) (-.001628,0.0) (.17853,0.0) (.37665,0.0)"
"(.37665,0.0) (.17853,0.0) (-.001628,0.0) (-.040609,0.0)"
}
```

For complex states, the syntax for the default value is (*real*, *imag*) where *real* and *imag* evaluate to integers or floats.

The *precision* state is used to give the precision of fixed-point values. These values may be other states or may be internal to the star. The default can be specified in either of two ways:

Method 1 : As a string like "3.2", or more generally "*m.n*", where *m* is the number of integer bits (to the left of the binary point) and *n* is the number of fractional bits (to the right of the binary point). Thus length is *m* + *n*.

Method 2: A string like "24/32" which means 24 fraction bits from a total length of 32. This format is often more convenient because the word length often remains constant while the number of fraction bits changes with the normalization being used.

In both cases, the sign bit counts as one of the integer bits, so this number must be at least one.

For *enum* states, the default value may only be one of the values listed in the *enumlist* field, in quotes ("value").

For *filename* states, the default value is the name of a file.

Units, such as MHz, msec, etc., can also be used in the default field, e.g., {"3 msec"}. However, this is not recommended since the value will not be displayed properly on the Schematic when the user changes the Units Scale Factors from the Options > Preferences > Units/Scale tab.

Expressions can also be used to set the default value. For example, the default field of a float state can be {"sin(0.3)"} or the default field of a complex state can be {"polar(3.2,

1.13)"}.

An example of the default field for each type is shown in the following table.

- The *desc* (or *descriptor*) field, which is optional but highly recommended, attaches a descriptor to the state. The same formatting options are available as with the star descriptor.
- The *attributes* field (optional) specifies state attributes. At present, four attributes are defined for all states: *A_CONSTANT*, *A_SETTABLE*, *A_SWEEPABLE*, and *A_SCHEMDISPLAY* (along with their complements *A_NONCONSTANT*, *A_NONSETTABLE*, *A_NONSWEEPABLE*, and *A_NOSCHEMDISPLAY*). If a state has the *A_CONSTANT* attribute, then its value is not modified by the run-time code in the star (it is up to you as the star writer to ensure that this condition is satisfied).

Default Fields

Type	Example Default
int	default {3}
float	default {3}
fix	default {1.25}
complex	default {"(1.25, 2.5)"}
string	default {"string value"}
precision	default {"24/32"}
enum	default {"value 1"}
filename	default {"/user/abc/xyz/lmn.txt"}
intarray	default {1, 2, 4, 7, 9}
floatarray	default {1.25, 3.50, 6.75}
complexarray	default {"(1.25, 2.5) (2.4, -2.3) (-1.2, -2.2)"}
stringarray	default {"Button1" "Button 2"}

States with the *A_NONCONSTANT* attribute may change when the star is run. If a state has the *A_SETTABLE* attribute, then user interfaces will enable the user to enter values for this state. States without this attribute are not presented to the user; such states always start with their default values as the initial value.

If a state has the *A_SWEEPABLE* attribute, its value can be swept and/or optimized using the appropriate simulation controllers. On the other hand, the *A_NONSWEEPABLE* attribute does not allow sweeping and/or optimizing the state's value.

The *A_SCHEMDISPLAY* attribute is only used for states that are *A_SETTABLE*. If a state has the *A_SCHEMDISPLAY* and *A_SETTABLE* attributes set, the state will be shown on the ADS schematic. If a state has the *A_NOSCHEMDISPLAY* and *A_SETTABLE* attributes set, the state will only be shown in the Edit Component Parameter dialog box.

Note that of all the attributes only *A_SCHEMDISPLAY* and *A_NOSCHEMDISPLAY* can be modified by the user for a specific instance of a component. This is done in the Edit Component Parameters dialog box by setting the "display parameter on schematic" flag appropriately.

If no attributes are specified, the default is

A_CONSTANT|A_SETTABLE|A_SCHEMDISPLAY|A_SWEEPABLE. Thus, in the above example, the *attributes* directive is unnecessary.

- The *units* field (optional) identifies the set of dimensional scale factors to be associated with this state at the schematic level in Advanced Design System. By default, the value of this field is UNITLESS_UNIT, which results in no scale factor association at the schematic level. Other unit options are shown in the following table.

Unit Options

Option	Function
FREQUENCY_UNIT	Results in use of frequency unit scale factors at the schematic level (GHz, MHz, etc.). The state in the code will receive a value in terms of Hz.
TIME_UNIT	Results in use of time unit scale factors at the schematic level (usec, msec, etc.). The state in the code will receive a value in terms of sec.
ANGLE_UNIT	Results in use of degree angle units at the schematic level. The state in the code will receive a value in terms of degree.
POWER_UNIT	Results in use of power scale and conversion factors at the schematic level (W, mW, dBm, dBW, etc.). The state in the code will receive a value in terms of W.
DISTANCE_UNIT	Results in use of distance unit scale factors at the schematic level (m, km, mile, etc.). The state in the code will reveal a value in terms of m.
LENGTH_UNIT	Results in use of length scale and conversion factors at the schematic level (m, mm, cm, in, ft, etc.). The state in the code will receive a value in terms of m.
RESISTANCE_UNIT	Results in use of resistance scale factors at the schematic level (Ohm, KOhm, MOhm, etc.). The state in the code will receive a value in terms of Ohm.
CAPACITANCE_UNIT	Results in use of capacitance scale factors at the schematic level (F, uF, pF, etc.). The state in the code will receive a value in terms of F.
INDUCTANCE_UNIT	Results in use of inductance scale factors at the schematic level (H, mH, uH, etc.). The state in the code will receive a value in terms of H.
CONDUCTANCE_UNIT	Results in use of conductance scale factors at the schematic level (S, mS, uS, etc.). The state in the code will receive a value in terms of S.
VOLTAGE_UNIT	Results in use of voltage scale factors at the schematic level (V, mV, uV, etc.). The state in the code will receive a value in terms of V.
CURRENT_UNIT	Results in use of current scale factors at the schematic level (A, mA, uA, etc.). The state in the code will receive a value in terms of H.
DB_GAIN_UNIT	Results in use of dB gain scale factors at the schematic level. The state in the code will receive a value in terms of dB.
TEMPERATURE_UNIT	Results in use of Celsius temperature scale factors at the schematic level. The state in the code will receive a value in terms of Celsius.

- The *enumlist* field is required when the state type is *enum*. The *enumlist* field is a comma separated list of strings.

enumlist {value 1, value 2, value 3}

Quotes around strings are optional. Spaces and other non-alphanumeric characters can be used. However, when referencing an *enum* value in the

code for the star, all non-alphanumeric characters must be replaced with an underbar (_). For example, "value 1" should be referenced as "value_1".

- The *enumlabels* field is optional and available for use only when the state type is *enum*. The *enumlabels* field contains name abbreviations for each *enumlist* value. The alternate names are for use only at the schematic level in Advanced Design System as a short mnemonic for the full enumeration value.

`enumlabels {v1, v2, v3}`

The label *v1* is used only as an abbreviation for *value 1*.

Two very commonly used enums are predefined: *query* with *enumlist* {NO, YES} and *boolean* with *enumlist* {FALSE, TRUE}. Below is an example showing their use:

```
state {
  name {periodic}
  type {query}
  default {YES}
}
```

- Multiple defaults parameter with Enum is a feature that enables developers to define non-Enum parameters, such as floating point and integer values, to provide multiple predefined values and user editable values.

Keywords: *type*, *enumlist* and *enumlabels* are used to correctly define a parameter with multiple defaults. The following shows a simple example.

```
defstar {
  name {MyTest}
  domain {TSDF}
  desc { My Constant output }
  vendor { AgilentEEsof }
  location { Numeric, Sources }
  output {
    name {output}
    type {float}
  }
  defstate {
    name {FCarrier}
    type { float }
    enumlist { ARFCN_1_10, ARFCN_2_20, ARFCN_3_30 }
    enumlabels { 10 MHz, 20 MHz, 30 MHz }
    default { ARFCN_2_20 }
    units { FREQUENCY_UNIT }
    desc { value }
  }
}
go {
  output%0 << double(FCarrier);
}
```

where:

type is used to define the type of the parameter.

enumlist is used to define the list of names associated to the default values.

enumlabels is used to list the values.

Both *enumlist* and *enumlabels* use commas to separate individual items.

- The filename state is just like a string state, except that the dialog box for a filename state can bring up a file browser for file selection. The extensions field can be specified for a filename state to list the valid extensions for the selected file. From a pull-down menu in the dialog box, you can select the files with certain extensions from the extensions field to be listed in the browser. If extensions is not specified or

is empty, the default extension used to list files in the browser is txt.

Mechanisms for accessing and updating states in C++ methods associated with a star are explained in the following list of keywords and in the sections [States](#) and [Array States](#).

input, output, inmulti, outmulti

These optional items are used to define a porthole, which may be an input, output porthole or an input, output multiporthole. Bidirectional ports are not supported. Like *state*, it contains subitems. The following is an example:

```
input {
  name {signalIn}
  type {complex}
  numtokens {2}
  desc {A complex input that consumes 2 input particles.}
}
```

Here, *name* specifies the porthole name. This is a required item.

The keyword *type* specifies the particle type. The scalar types are *int*, *float*, *fix*, *complex*, *message*, or *anytype*. Again, case does not matter for the type value. The matrix types are *int_matrix*, *float_matrix*, *complex_matrix*, and *fix_matrix*. The *type* item may be omitted. The default type is *anytype*. For more information on all of these, refer to *Data Types for Model Builders* (modbuild). The *numtokens* keyword (it may also be written *num* or *numTokens*) specifies the number of tokens consumed or produced on each firing of the star. This only makes sense for certain domains (SDF and TSDF). In such domains, if the item is omitted, a value of one is used. For stars where this number depends on the value of a state, it is preferable to leave out the *numtokens* specification and to have the *setup* method set the number of tokens. (In the SDF and TSDF domains, this is accomplished with the *setSDFParams* method.) This item is used primarily in the SDF and TSDF domains, and is discussed further in the documentation of these domains.

There is an alternative syntax for the *type* field of a porthole. This syntax is used in connection with *ANYTYPE* to specify a link between the types of two portholes. The syntax is:

```
type {= name }
```

where *name* is the name of another porthole. This indicates that this porthole inherits its type from the specified porthole. For example, here is a portion of the definition of the SDF *Fork* star:

```
input {
  name{input}
  type{ANYTYPE}
}
outmulti {
  name{output}
  type{= input}
  desc{type is inherited from the input}
```

}

constructor

This optional item enables the user to specify extra C++ code to be executed in the constructor for the class. This code will be executed after any automatically generated code in the constructor that initializes portholes, states, etc. The syntax is:

```
constructor {body}
```

where *body* is a piece of C++ code. It can be of any length. Note that the constructor is invoked only when the class is first instantiated; actions that must be performed before every simulation run should appear in the *setup* or *begin* methods, not the constructor.

conscalls

With this optional item, you might have data members in your star that have constructors requiring arguments. These members would be added by using the *public*, *private*, or *protected* keywords. If you have such members, the *conscalls* keyword provides a mechanism for passing arguments to the constructors of those members. Simply list the names of the members followed by the list of constructor arguments for each, separated by commas if there is more than one. The syntax is:

```
conscalls {member1(arglist), member2(arglist)}
```

Note that *member1*, and *member2* should have been previously defined in a *public*, *private*, or *protected* section. (See the subsequent descriptions of these keywords.)

destructor

This optional item inserts code into the destructor for the class. The syntax is:

```
destructor {body}
```

You generally need a destructor only if you allocate memory in the constructor, *begin* method, or *setup* method; termination functions that happen with every run should appear in the *wrapup* function. (*Wrapup* is not called if an error occurs. See subsequent description of the *wrapup* keyword.) The optional keyword *inline* may appear before *destructor*. If so, the destructor function definition appears inline, in the header file. Since the destructor for all stars is virtual, this is only a win when the star is used as a base for derivation.

setup

This optional item defines the *setup* method, which is called every time the simulation is started, *before* any compile-time scheduling is performed. The syntax is:

```
setup {body}
```

The optional keyword *inline* may appear before the *setup* keyword. It is common for this method to set parameters of input and output portholes, and to initialize states. For an explanation of the code syntax for doing this, refer to the section, [Reading Inputs and Writing Outputs](#). In some domains, with some targets, the *setup* method may be called more than once during initiation. You must keep this in mind if you use it to allocate or initialize memory.

begin

This optional item defines the *begin* method, which is called every time the simulation is started, but *after* the scheduler *setup* method is called (that is, after any compile-time scheduling is performed). The syntax is:

```
begin {body}
```

This method can be used to allocate and initialize memory. It is especially useful when data structures are shared across multiple instances of a star. It is always called exactly once when a simulation is started.

go

This optional item defines the action taken by the star when it is fired. The syntax is:

```
go {body}
```

The optional keyword *inline* may appear before the *go* keyword. The *go* method will typically read input particles and write outputs, and will be invoked many times during the course of a simulation. For an explanation of the code syntax for the body, refer to the section, [Reading Inputs and Writing Outputs](#).

wrapup

This optional item defines the *wrapup* method, which is called at the completion of a simulation. The syntax is:

```
wrapup {body}
```

The optional keyword *inline* may appear before the *wrapup* keyword. The *wrapup* method might typically display or store final state values. For an explanation of the code syntax

for doing this, refer to the section, [Reading Inputs and Writing Outputs](#). Note that the *wrapup* method is not invoked if an error occurs during execution. Thus, the *wrapup* method cannot be used reliably to free allocated memory. Instead, you should free memory from the previous run in the *setup* or *begin* method, prior to allocating new memory, and in the destructor.

public, protected, private

These optional items enable you to declare extra members for the class with the desired protection. The syntax is:

```
protkey {body}
```

where *protkey* is *public*, *protected*, or *private*. Example, from the *XMgraph* star:

```
protected {
    XGraph graph;
    double index;
}
```

This defines an instance of the class *XGraph*, defined in the ADS Ptolemy kernel, and a double-precision number. If any of the added members require arguments for their constructors, use the *conscalls* item to specify them.

ccinclude, hinclude

These optional items cause the *.cc* file, or the *.h* file, to *#include* extra files. A certain number of files are automatically included, when the preprocessor can determine that they are needed, so they do not need to be explicitly specified. The syntax is:

```
ccinclude {inclist}
hinclude {inclist}
```

where *inclist* is a comma-separated list of include files. Each filename must be surrounded either by quotation marks or by < and > (for system include files like <*math.h*>).

code

This optional item enables the user to specify a section of arbitrary C++ code. This code is inserted into the *.cc* file after the include files, but before everything else; it can be used to define static non-class functions, declare external variables, or anything else. The outermost pair of curly braces is stripped. The syntax is:

```
code {body}
```

header

This optional item enables the user to specify an arbitrary set of definitions that will appear in the header file. Everything between the curly braces is inserted into the *.h* file after the include files but before everything else. This can be used, for example, to define classes used by your star. The outermost pair of curly braces is stripped.

method

This optional item provides a fully general way to specify an additional method for the class of star that is being defined. Here is an example:

```
virtual method {
  name {exec}
  access {protected}
  arglist {"(const char* extraOpts)"}
  type {void}
  code {
    // code for the exec method goes here
  }
}
```

An optional function type specification may appear before the *method* keyword, which must be one of the following:

```
virtual
inline
pure
pure virtual
inline virtual
```

The *virtual* keyword makes a virtual member function. If the *pure virtual* keyword is given, a pure virtual member function is declared (there must be no *code* item in this case). The function type *pure* is a synonym for *pure virtual*. The *inline* function type declares the function to be inline.

The following are the *method* subitems:

- *name* (Name of the method; required item).
- *access* (Level of access for the method, one of *public*, *protected*, or *private*. If the item is omitted, *protected* is assumed).
- *arglist* (Argument list, including the outermost parentheses, for the method as a quoted string. If this is omitted, the method has no arguments.)
- *type* (Return type of the method. If the return type is not a single identifier, you must put quotes around it. If this is omitted, the return type is *void*; no value is returned).
- *code* (C-code that implements the method. This is a required item, unless the *pure* keyword appears, in which case this item *cannot* appear).

vendor

This optional item provides a way of specifying the source of a given star. For example, {Agilent EEsof} declares that Agilent EEsof is the provider of the model. This field is displayed in the Advanced Design System browser.

Writing C++ Code for Stars

This section assumes a knowledge of the C++ language. For those new to the language, we recommend "The C++ Programming Language, Third Edition," by Bjarne Stroustrup (from Addison-Wesley).

C++ code segments are an important part of any star definition. They can appear in the *setup*, *begin*, *go*, *wrapup*, *constructor*, *destructor*, *exectime*, *header*, *code*, and *method* directives in the ADS Ptolemy preprocessor. These directives all include a body of arbitrary C++ code, enclosed by curly braces, " {" and " }". In all but the *code* and *header* directives, the C++ code between braces defines the body of a method of the star class. Methods can access any member of the class, including portholes (for input and output), states, and members defined with the *public*, *protected*, and *private* directives.

The Structure of an ADS Ptolemy Star

In general, the task of an ADS Ptolemy star is to receive input particles and produce output particles. In addition, there may be side effects (reading or writing files, displaying graphs, or even updating shared data structures). As for all C++ objects, the constructor is called when the star is created, and the destructor is called when it is destroyed. In addition, the *setup* and *begin* methods, if any, are called every time a new simulation run is started, the *go* method (which always exists, except for stars like *BlackHole* and *Null* that do nothing) is called each time a star is executed, and the *wrapup* method is called after the simulation run completes without errors.

Messaging Guidelines for Star .pl Files

This section provides guidelines for creating messages for display in the Advanced Design System Status window (as is done for all Agilent EEsof stars). Messages are needed to communicate status, warning, and error information. Examples of messages used in star files can be seen in the *pl* files located at *\$HPEESOF_DIR/doc/sp_files*. All messages use methods from the ADS Ptolemy Error class and have the general form:

```
Error::<type>(<argument_list>);
where
<type> = message for communicating status information
<type> = warn for communicating warning information
```


< *type* > = *initialization* for communicating error during simulation initialization and setup
 < *type* > = *abortRun* for communicating an error that will end the simulation
 < *argument_list* > = argument list for the specific Error class method

An additional method for state range error reporting has the form:

```
<state_name>::rangeError(<argument_list>);
where
< state_name > = name of the state
< argument_list > = argument list for the specific Error class method
```

Status Messages

Status messages do not have any specific starting token in the argument list and allow the simulation to conclude. Status messages are typically used in the *setup()* and *go()* methods. Status messages can be created using the *Error::message()* methods. These methods have the following prototypes:

```
Error::message(const char *, const char * = 0, const char * = 0);
Error::message(const NamedObj&, const char *, const char * = 0, const char *
= 0);
```

Where possible, the second method should be used so that the name of the *NamedObj* can be displayed along with the message. The named object can be **this* to mean the current star instance.

Warning Messages

Warnings have their messages automatically prefixed with the token *Warning:*, and allow the simulation to conclude. Warning messages are typically used in the *setup()* and *go()* methods. They can be created using the *Error::warn()* methods. These methods have the following prototypes:

```
Error::warn(const char *, const char * = 0, const char * = 0);
Error::warn(const NamedObj&, const char *, const char * = 0, const char * =
0);
```

Where possible, the second method should be used so that the name of the *NamedObj* can be displayed along with the message. The named object can be **this* to mean the current star instance.

Error Messages

Errors have their messages automatically prefixed with the token *ERROR:*, and result in

stopping the simulation. Error messages used before the *go()* method are called during a simulation and should not cause the simulation to stop until after the completion of the simulation initialization and setup.

During the *setup()* method, state values should be checked for any value range error. If an error exists in the state value, the *State::rangeError()* method should be used:

```
<state_name>.rangeError(const char *);
```

where the *const char ** is a string that defines the required state range.

Examples:

```
FCarrier.rangeError(">= 0.0");
Top.rangeError("> Bottom");
```

If an error other than this state range error occurs during the initialization process, the *Error::initialization()* methods should be used. These methods have the following prototypes:

```
Error::initialization(const char *, const char * = 0, const char * = 0);
Error::initialization(const NamedObj&, const char *, const char * = 0, const char *
= 0);
```

Where possible, the second method should be used so that the name of the *NamedObj* can be displayed along with the message. The named object can be **this* to mean the current star instance.

Error messages used in the *go()* method will not cause the simulation to stop until after the current *go()* method is complete.

These error messages can be created using the *Error::abortRun()* methods. These methods have the following prototypes:

```
Error::abortRun(const char *, const char * = 0, const char * = 0);
Error::abortRun(const NamedObj&, const char *, const char * = 0, const char *
= 0);
```

Where possible, the second method should be used so that the name of the *NamedObj* can be displayed along with the message. The named object can be **this* to mean the current star instance.

Reading Inputs and Writing Outputs

The precise mechanism for references to input and output portholes depends somewhat on the domain. This is because stars in the domain *XXX* use objects of class *InXXXPort* and *OutXXXPort* (derived from *PortHole*) for input and output, respectively. The examples used here are for the SDF (or TSDF) domain. See the appropriate domain section ___ for variations that apply to other domains.

PortHoles and Particles

In the SDF (TSDF) domain, normal inputs and outputs become members of type *InSDFPort* (*InTSDFPort*) and *OutSDFPort* (*OutTSDFPort*) after the preprocessor is finished. These are derived from base class *PortHole*. For example, given the following directive in the *defstar* of an *SDF* (*TSDF*) star,

```
input {
    name {in}
    type {float}
}
```

a member named *in*, of type *InSDFPort* (*InTSDFPort*), will become part of the star.

We are not usually interested in directly accessing these porthole classes, but rather wish to read or write data through the portholes. All data passing through a porthole is derived from base class *Particle*. Each particle contains data of the type specified in the *type* subdirective of the *input* or *output* directive.

The operator `%` operating on a porthole returns a reference to a particle. Consider the following example:

```
go {
    Particle& currentSample = in%0;
    Particle& pastSample = in%1;
    ...
}
```

The right-hand argument to the `%` operator specifies the delay of the access. A zero always means the most recent particle. A one means the particle arriving just before the most recent particle. This also applies to outputs. Given an output named *out*, the particles that are read from *in* can be written to *out* in the same order as follows:

```
go {
    ...
    out%1 = pastSample;
    out%0 = currentSample;
}
```

This works because *out%n* returns a reference to a particle, and hence can accept an assignment. The assignment operator for the class *Particle* is overloaded to make a copy of the data field of the particle.

Operating directly on class *Particle*, as in the above examples, is useful for writing stars that accept *anytype* of input. The operations don't need to concern themselves with the type of data contained by the particle. But it is far more common to operate numerically on the data carried by a particle. This can be done using a cast to a compatible type. For example, since *in* above is of type *float*, its data can be accessed as follows:

```
go {
```

```
Particle& currentSample = in%0;
double value = double(currentSample);
...
}
```

or more concisely,

```
go {
    double value = double(in%0);
    ...
}
```

The expression *double(in%0)* can be used anywhere that a double can be used. In many contexts, where there is no ambiguity, the conversion operator can be omitted:

```
double value = in%0;
```

However, since conversion operators are defined to convert particles to several types, it is often necessary to indicate precisely which type conversion is desired.

To write data to an output porthole, note that the right-hand side of the assignment operator should be of type *Particle*, as shown in the above example. An operator *<<* is defined for particle classes to make this more convenient. Consider the following example:

```
go {
    float t;
    t = some value to be sent to the output
    out%0 << t;
}
```

Note the distinction between the *<<* operator and the assignment operator. The latter operator copies *Particles*, the former operator loads data into particles. The type of the right-side operand of *<<* may be *int*, *float*, *double*, *Fix*, *Complex* or *Envelope*. Note that the *Envelope* data class includes the matrix data types. The appropriate type conversion will be performed. For more information on the *Envelope* and *Message* types, refer to *Data Types for Model Builders* (modbuild).

SDF (TSDF) PortHole Parameters

In the preceding example, where *in%1* was referenced, some special action is required to tell ADS Ptolemy that past input particles are to be saved. Special action is also required to tell the *SDF* (*TSDF*) scheduler how many particles will be consumed at each input and produced at each output when a star fires. This information can be provided through a call to *setSDFParams* (*setTSDFParams*) in the method. This has the syntax:

```
setup {
    name.setSDFParams(multiplicity, past)
}
```

where *name* is the name of the input or output porthole, *multiplicity* is the number of particles consumed or produced, and *past* is the maximum value that *offset* can take in any expression of the form *name%offset*. For example, if the *go {{}}* method references *name%0* and *name%1*, then *past* would have to be at least one. It is zero by default.

Multiple PortHoles

Sometimes a star should be defined with n input portholes or n output portholes, where n is variable. This is supported by the class *MultiPortHole*, and its derived classes. An object of this class has a sequential list of *PortHoles*. For *SDF* (*TSDF*), we have the specialized derived class *MultiInSDFPort* (*MultiInTSDFPort*), which contains *InSDFPorts* (*InTSDFPorts*) and *MultiOutSDFPort* (*MultiOutTSDFPort*), which contains *OutSDFPorts* (*OutTSDFPorts*).

Defining a multiple porthole is easy, as illustrated below:

```
defstar {
    ...
    inmulti {
        name {input_name}
        type {input_type}
    }
    outmulti {
        name {output_name}
        type {output_type}
    }
    ...
}
```

To successively access individual portholes in a *MultiPortHole*, the *MPHIter* iterator class should be used. Consider the following code segment from the definition of the *SDF Fork* (*TSDF Fork*) star:

```
input {
    name{input}
    type{ANYTYPE}
}
outmulti {
    name{output}
    type{= input}
}
go {
    MPHIter nextp(output);
    PortHole* p;
    while ((p = nextp++) != 0)
        (*p)%0 = input%0;
}
```

A single input porthole supplies a particle that gets copied to any number of output portholes. The *type* of the output *MultiPortHole* is inherited from the type of the input. The first line of the *go* method creates an *MPHIter* iterator called *nextp*, initialized to point to portholes in *output*. The *++* operator on the iterator returns a pointer to the next porthole in the list, until there are no more portholes, at which time it returns *NULL*. So the *while* construct steps through all output portholes, copying the input particle data to each one.

Consider another example, taken from the *SDF Add* star:

```
inmulti {
    name {input}
    type {float}
```

```

}
output {
    name {output}
    type {float}
}
go {
    MPHIter nexti(input);
    PortHole *p;
    double sum = 0.0;
    while ((p = nexti++) != 0)
        sum += double((*p)%0);
    output%0 << sum;
}

```

Again, an *MPHIter* iterator named *nexti* is created and used to access the inputs.

The *numberPorts* method of class *MultiPortHole*, which returns the number of ports, is occasionally useful. This is called simply as *portname.numberPorts()*, and it returns an *int*.

Type Conversion

The type conversion operators and *<<* operators are defined as virtual methods in the base class *Particle*. There are never really objects of class *Particle* in the system. Instead, there are objects of class *IntParticle*, *FloatParticle*, *ComplexParticle*, and *FixParticle*, which hold data of type *int*, *double* (not float), *Complex*, and *Fix*, respectively. (There are also *MessageParticle* and a variety of matrix particles). The conversion and loading operators are designed to do the right thing when an attempt is made to convert between mismatched types.

Clearly we can convert an *int* to a *double* or *Complex*, or a *double* to a *Complex*, with no loss of information. Attempts to convert in the opposite direction work as follows: conversion of a *Complex* to a *double* produces the magnitude of the complex number. Conversion of a *double* to an *int* produces the greatest integer that is less than or equal to the *double* value. There are also operators to convert to or from *float* and *Fix*. Each particle also has a virtual *print* method, so a star that writes particles to a file can accept *anytype*.

States

A state is defined by the *state* directive. The star can use a state to store data values, remembering them from one invocation to another. States differ from ordinary members of the star (defined by the *public*, *protected*, and *private* directives) in that they have a name, and can be accessed from outside the star in systematic ways. For instance, the Advanced Design System design environment enables you to set any state with the *A_SETTABLE* attribute to some value prior to a run; this is done via the on schematic value entry or the Edit Component dialog. The state attributes are set in the *state* directive.

A state may be modified by the star code during a run. To mark a state as one that gets modified during a run, use the attribute *A_NONCONSTANT*. There is currently no mechanism for checking the correctness of these attributes.

All states are derived from the base class *State*, defined in the ADS Ptolemy kernel. The derived state classes currently defined in the kernel are *FloatState*, *IntState*, *ComplexState*, *StringState*, *FileNameState*, *FloatArrayState*, *IntArrayState*, *ComplexArrayState*, *StringArrayState*, *EnumerationState*, and *PrecisionState*.

A state can be used in a star method in the same way as the corresponding predefined data types. As an example, suppose the star definition contains the following directive:

```
state {
    name {myState}
    type {float}
    default {1.0}
    descriptor {Gain parameter.}
}
```

This will define a member of class *FloatState* with default value 1.0. No attributes are defined, so *A_CONSTANT* and *A_SETTABLE*, the default attributes, are assumed. To use the value of a state, it should be cast to type *double*, either explicitly by the programmer or implicitly by the context. For example, the value of this state can be accessed in the *go* method as follows:

```
go {
    output%0 << double(myState) * double(input%0);
}
```

The references to input and output are explained above. The reference to *myState* has an explicit cast to *double* ; this cast is defined in the *FloatState* class. Similarly, a cast to *int* is available for *IntState*, to *Complex* for *ComplexState*, and to *const char** for *Stringstate*). In principle, it is possible to rely on the compiler to automatically invoke this cast. However, note the following warning.

Explicit casting should be used whenever a state is used in an expression. For example, from the setup method of the *SDFChop* star, in which *use_past_inputs* is an integer state,

```
if (int(use_past_inputs))
    input.setSDFParams(int(nread),int(nread)+int(offset)-1);
else
    input.setSDFParams(int(nread),int(nread)-1);
```

Note that the type *Complex* is not a fundamental part of C++. We have implemented a subset of the *Complex* class as defined by several library vendors. We use our own version for maximum portability. Using the *ComplexState* class automatically ensures the inclusion of the appropriate header files. A member of the *Complex* class can be initialized and operated upon any number of ways. For details, refer to the section, *The Complex Data Type* (modbuild) in *Data Types for Model Builders* (modbuild).

A state may be updated by ordinary assignment in C++, as in the following lines:

```
double t = expression;
myState = t;
```

This works because the *FloatState* class definition has overloaded the assignment operator

(=) to set its value from a *double*. Similarly, an *IntState* can be set from an *int*, and a *StringState* can be set from a *char** or *const char**.

Array States

The *ArrayState* classes (*FloatArrayState*, *IntArrayState* and *ComplexArrayState*) are used to store data arrays. For example,

```
state {
  name {taps}
  type {FloatArray}
  default {"0.0 0.0 0.0 0.0"}
  descriptor {An array of length four.}
```

defines an array of type *double* with dimension four, with each element initialized to zero. Quotes must surround the initial values. Alternatively, you can specify a file name with the prefix <. If you have a file named *foo* that contains the default values for an array state, you can write:

```
default {"< foo"}
```

where the file *foo* must be located in the current workspace data subdirectory. If not in the subdirectory, then the filename must include the full directory path as a prefix. For instance:

```
default {"< ~/user_name/directory/foo"}
```

The format of the file is also a sequence of data separated by spaces (or new lines, tabs, or commas). File input can be combined with direct data input as in:

```
default {"< foo 2.0"}
default {"0.5 < foo < bar"}
```

A repeat notation is also supported for *ArrayState* objects: the two value strings

```
default {"1.0 [5]"}
default {"1.0 1.0 1.0 1.0 1.0"}
```

are equivalent. Any integer expression may appear inside the brackets *[]*. The number of elements in an *ArrayState* can be determined by calling its *size* method. The size is not specified explicitly, but is calculated by scanning the default value.

As an example of how to access the elements of an *ArrayState*, suppose *fState* is a *FloatState* and *aState* is a *FloatArrayState*. The access points, like those in the following lines, are routine:

```
fState = aState[1] + 0.5;
aState[1] = (double)fState * 10.0;
aState[0] = (double)fState * aState[2];
```


For a more complete example of the use of *FloatArrayState*, consider the *FIR* star defined below. Note that this is a simplified version of the SDF *FIR* star and does not permit interpolation or decimation.

```
defstar {
  name {FIR}
  domain {SDF}
  desc {
A Finite Impulse Response (FIR) filter.
  }
  input {
    name {signalIn}
    type {float}
  }
  output {
    name {signalOut}
    type {float}
  }
  state {
    name {taps}
    type {floatarray}
    default {      "-.04 -.001 .17 .37 .37 .17 -.0018 -.04" }
    desc {Filter tap values.}
  }
  setup {
    // tell the PortHole the maximum delay we will use
    signalIn.setSDFParams(1, taps.size() - 1);
  }
  go {
    double out = 0.0;
    for (int i = 0; i < taps.size(); i++)
      out += taps[i] * double(signalIn%i);
    signalOut%0 << out;
  }
}
```

Notice the *setup* method; this is necessary to allocate a buffer in the input *PortHole* large enough to hold the particles that are accessed in the *go* method. Notice also the use of the *size* method of the *FloatArrayState*.

Modifying PortHoles and States in Derived Classes

When one star is derived from another, it inherits all the states of the base class star. Sometimes we want to modify some aspect of the behavior of a base class state in the derived class. This is done by placing calls to member functions of the state in the constructor of the derived star. Useful functions include *setInitValue* to change the default value, and *setAttributes* and *clearAttributes* to modify attributes.

When creating new stars derived from stars already in the system, you will often also wish to customize them by adding new ports or states. In addition, you may wish to remove ports or states. Although, strictly speaking, you cannot do this, you can achieve the desired effect by simply hiding them from the user.

The following code will hide a particular state named *statename* from the user:

```
constructor {
```

```

    statename.clearAttributes(A_SETTABLE);
}

```

Thus, when the user observes the available states for this star in the Advanced Design System design environment, *statename* will not appear as one of the star parameters. Of course, the state can still be set and used within the code defining the star.

The same effect can be achieved with outputs or inputs. For instance, given an output named *output*, you can use the following code:

```

constructor {
    output.setAttributes(P_HIDDEN);
}

```

This means that when you create an icon for this star, no terminal appears for this port. This is most useful when *output* is a multiporthole, because there will then be zero instances of the individual portholes.

This technique can also be used to hide individual portholes. However, it must be used with caution because the porthole still remain. Most domains do not allow disconnected portholes, and will flag an error. You can explicitly connect the port within the body of the star.

Writing Timed Components

Writing Timed components using hpeesoflang is almost identical to writing any other star. Following are the primary points of distinction:

- Receiving Timed data
To receive the data field of the Timed data via input TSDFPortHole, use the following method:

```
Complex InTSDFPort::getIQData(int n)
```

where *n* is the current value of the input stream. For example, if the Timed input port is named *in*, then

```
in.getIQData(0)
```

returns a complex number, which is the current I and Q members of the Timed particle.

Similarly, the methods

```
int InTSDFPort::getFlavor(int)
```

and

```
double TSDFPortHole::getCarrierFrequency(int)
```

return the Flavor and Fc associated with incoming Timed particle.

- Sending data
As described in the preceding section [Reading Inputs and Writing Outputs](#), the

operator << is used to load the output port with Timed data. For example, given the Timed ports out1, out2, the following will output Baseband and ComplexEnv flavor Timed data at out1 and out2 ports. Note that the other attributes of Timed particle are set by the engine.

```
go{
double x;
Complex z;
.....
out1%0 << x;
out2%0 << z;
}
```

- Fc propagation

When a TSDF star is *changing (or re-setting)* the carrier frequency F_c , a TSDFStar method should be used in the star setup as follows:

TSDFStar::propagateFc(double fc)

If this method is not explicitly used, the virtual method is used, which sets the output carrier frequency equal to the maximum input carrier frequency.

Example of Writing Timed Components

```
method {
name {propagateFc}
access {protected}
arglist {"(double *fcin)"}
type {void}
code {
output.setCarrierFrequency(dummy);
}
}
```

- TStep propagation

When a TSDF star is *changing or re-setting* the TStep (for example a source), a TSDFStar method should be used in the star setup, as follows:

TSDFPortHole::setTimeStep(double timestep)

- ComplexToTimed Converter example

```
defstar {
name {CxToTimed}
domain {TSDF}
desc {Converts a Complex signal to Timed. Given the
complex number (a+bj) at input, the output is a
ComplexEnv Timed signal
{(I + jQ),fc} where I=a, Q=b and fc is a parameter.}
copyright {Copyright (c) Agilent Technologies 2000}
attributes {S_HP}
location {Signal Converters}
input {
name {input}
type {Complex}}
output {
name {output}
type {timed}}
defstate {
name {TStep}
type {float}
default {0.0}
```

User-Defined Models

```
desc {Output time step}
units {TIME_UNIT}
attributes {A_SETTABLE|A_NONCONSTANT}}

defstate {
    name {FCarrier}
    type {float}
    default {-1.0}
    desc {Output Carrier frequency}
    units {FREQUENCY_UNIT}
    attributes {A_SETTABLE|A_NONCONSTANT}}

setup {
    if (double(TStep) < 0.)
        TStep.rangeError(">= 0");
    output.setTimeStep((double)TStep);
    output.setCarrierFrequency((double)FCarrier);
}

// for Fc propagation, overriding the virtual TSDFStar::propagateFc()
method {
    name {propagateFc}
    access {protected}
    arglist {"(double *fcin)"}
    type {void}
    code {
        output.setCarrierFrequency(dummy);}
    }

    go {
        output%0 << (Complex)(input%0);
    }
}
```

Programming Examples

The following star has no inputs, just an output. The source star generates a linearly increasing or decreasing sequence of float particles on its output. The state *value* is initialized to define the value of the first *output*. Each time the star *go* method fires, the *value* state is updated to store the next *output* value. Hence, the attributes of the *value* state are set so that the state can be overwritten by the star's methods. By default, the star will generate the output sequence 0.0, 1.0, 2.0, etc.

```
defstar {
    name {Ramp}
    domain {SDF}
    desc {
        Generates a ramp signal, starting at "value" (default 0)
        with step size "step" (default 1).
    }
    output {
        name {output}
        type {float}
    }
    state {
        name {step}
        type {float}
        default {1.0}
        desc {Increment from one sample to the next.}
    }
    state {
        name {value}
        type {float}
        default {0.0}
        desc {Initial (or latest) value output by Ramp.}
        attributes {A_SETTABLE|A_NONCONSTANT}
    }
    go {
        double t = double(value);
```

```

        output%0 << t;
        t += step;
        value = t;
    }
}

```

The next example is the *Gain* star, which multiplies its input by a constant and outputs the result:

```

defstar {
    name { {anchor:1105762:index:Gain (SDF block)}Gain}
    domain {SDF}
    desc {Amplifier: output is input times "gain" (default 1.0).}
    input {
        name {input}
        type {float}
    }
    output {
        name {output}
        type {float}
    }
    state {
        name {gain}
        type {float}
        default {"1.0"}
        desc {Gain of the star.}
    }
    go {
        output%0 << double(gain) * double(input%0);
    }
}

```

The following example of the *Printer* star illustrates multiple inputs, *ANYTYPE* inputs, and the use of the *print* method of the *Particle* class.

```

defstar {
    name {{anchor:1105765:index: Printer (SDF block)}Printer}
    domain {SDF}
    inmulti {
        name {input}
        type {ANYTYPE}
    }
    state {
        name {fileName}
        type {string}
        default {"<cout>"}
        desc {Filename for output.}
    }
    hinclude {"pt_fstream.h"}
    protected {
        pt_ofstream *p_out;
    }
    constructor {p_out = 0;}
    destructor {LOG_DEL; delete p_out;}
    setup {
        delete p_out;
        p_out = new pt_ofstream(fileName);
    }
    go {
        pt_ofstream& output = *p_out;
        MPHIter nexti(input);
        PortHole* p;
        while ((p = nexti++) != 0)
            output << ((*p)%0).print() << "t";
        output << "n";
    }
}

```

This star is polymorphic since it can operate on any type of input. Note that the default value of the output filename is `<cout>`, which causes the output to go to the standard output.

Preventing Memory Leaks in C++ Code

Memory leaks occur when new memory is allocated dynamically and never deallocated. In C programs, new memory is allocated by the *malloc* or *calloc* functions, and deallocated by the *free* function. In C++, new memory is usually allocated by the *new* operator and deallocated by the *delete* or the *delete []* operator. The problem with memory leaks is that they accumulate over time and, if left unchecked, may cripple or even crash a program. Agilent EEsof has taken extensive steps to eliminate memory leaks in the ADS Ptolemy software environment by implementing the following guidelines and by tracking memory leaks with Purify (a commercial tool from Pure Software, Inc.).

- One of the most common mistakes leading to memory leaks is applying the wrong *delete* operator. The *delete* operator should be used to free a single allocated class or data value, whereas the *delete []* operator should be used to free an array of data values. In C programming, the *free* function does not make this distinction.
- Another common mistake is overwriting a variable containing dynamic memory without freeing any existing memory first. For example, assume that *thestring* is a data member of a class, and in one of the methods (other than the constructor), there is the following statement:

```
thestring = new char[buflen];
```

This code should be

```
delete [] thestring;
thestring = new char[buflen];
```

Using *delete* is not necessary in a class' constructor because the data member would not have been previously allocated.

- In writing ADS Ptolemy stars, the *delete* operator should be applied to variables containing dynamic memory in both the star's setup and destructor methods. In the star's constructor method, the variables containing dynamic memory should be initialized to zero. By freeing memory in both the setup and destructor methods, one covers all possible cases of memory leaks during simulation. Deallocating memory in the setup method handles the case in which the user restarts a simulation, whereas deallocating memory in the destructor covers the case in which the user exits a simulation. This includes the cases that arise when error messages are generated.
- Another common mistake is not paying attention to the kinds of strings returned by functions. The function *savestring* returns a new string dynamically allocated and should be deleted when no longer used. The *expandPathName*, *tempFileName*, and *makeLower* functions return new strings, as does the *Target::writeFileName* method. Therefore, the strings returned by these routines should be deleted when they are no longer needed, and code such as

```
savestring(expandPathName(s))
```

is redundant and should be simplified to

```
expandPathName(s)
```

to avoid a memory leak due to not keeping track of the dynamic memory returned by the function *savestring*.

- Occasionally, dynamic memory is used when local memory could have been used instead. For example, if a variable is only used as a local variable inside a method or function, and the value of the local variable is not returned or passed to outside the method or function, then it is better to simply use local memory. For example, the sequence

```
char* localstring = new char[len + 1];
if (person == absent) return;
strcpy(localstring, otherstring);
delete [] localstring;
return;
```

could easily return without deallocating *localstring*. The code should be rewritten to implement either the *StringList* or *InfString* class; for example:

```
InfString localstring;
if (person == absent) return;
localstring = otherstring;
return;
```

Both *StringList* and *InfString* can manage the construction of strings of arbitrary size. When a function or method exits, the destructors of the *StringList* and *InfString* variables are automatically called, which deallocates their memory. Casts that convert *StringList* to a *const char** string and *InfString* to a *const char** or a *char** string are defined, so that instances of the *StringList* and *InfString* classes can be passed as is into routines that take character array (string) arguments. The following is a simple example of the function that builds an error message into a single string:

```
StringList sl = msg;
sl << file << ": " << sys_errlist[errno];
ErrAdd(sl);
```

The *errAdd* function takes a *const char** argument, so *sl* is automatically converted to a *const char** string by the C++ compiler.

Instead of using the new and delete operators, it is tempting to use constructs like:

```
char localstring[buflen + 1];
```

in which *buflen* is a variable. This is because the compiler will then automatically handle memory deallocation. Unfortunately, this syntax is a Gnu extension and is not portable to other C++ compilers. Instead, the *StringList* and *InfString* classes should be used, as in the previous example involving *localstring*.

Sometimes the return value from a routine that returns dynamic memory is not stored and, therefore, the pointer to the dynamic memory gets lost. This occurs, for

example, in nested function calls. Code such as

```
puts(savestring(s));
```

should instead be written as

```
const char* newstring = savestring(s);
puts(newstring);
delete [] newstring;
```

Several features in ADS Ptolemy, especially in the schedulers and targets, rely on the *hashstring* function, which returns dynamic memory. This dynamic memory, however, should not be deallocated because it may be reused by other calls to *hashstring*. It is the responsibility of the *hashstring* function to deallocate any memory it has allocated.

ADS Ptolemy pl File Template

The following is a pl file template:

```
adsptolemy Star coding template
defstar {
name {my_model}          // Limit length to one line 30 characters maximum
// No spaces. Only alpha-numeric characters and underbar.
// Name should be constructed as one or more concatenated
// word segments with each word segment beginning with a capital letter.
domain {SDF} // or TSDF
desc {my_model_name}     // Limit length to one line of 50 characters maximum
// This description should be a short phrase defining the star
// Do not use a period followed by a space; ". "
// The period followed by a space is recognized by adsptolemy
// as the end of the descriptions to be displayed in AEL
// A detailed model explanation should be placed in the
// explanation {} field
version {@(#) $Source:
/wlv/src/sp100/source/ptolemy/src/domains/sdf/stars/SDFmy_model.pl $
$Revision: 1.0 $ $Date: 1997/10/28 16:26:58 $}
// This version field to be changed as needed for HMS source code control by the user
author {Author's name}
acknowledge {arbitrary single line of text to acknowledge others}
location {my_model_library_location} // Name of Library used in the Schematic
attributes {S_USER} // or S_HIDDEN
derivedfrom {base_star_name} // Optional: delete if not used
copyright {
Copyright (c) Agilent Technologies 2000
All rights reserved.
}
explanation {my_model_explanation}
// Use as many lines as needed to describe the star, it's purpose,
// algorithm, application, references, or other information to
// document this component
// Define a defstate for each parameter
defstate {
name {my_state_name} // Limit length to one line 30 characters maximum
// No spaces. Only alpha-numeric characters and underbar.
// Name should be constructed as one or more word segments
// with each word segment beginning with a capital letter.
type {my_state_type} // Options: int, fix, float, complex, string, precision,
// intarray, fixarray, floatarray, complexarray, stringarray
// For the enum state, see the next defstate{} example
// For the filename state, see the following defstate{} example
```


User-Defined Models

```

default {my_state_default_value}
    // Example int:                1
    // Example float:              1.25
    // Example fix:                1.25
    // Example complex:            "(1.25, 2.5)"
    // Example string:              "my string"
    // Example precision:          2.14
    // Example intarray:           1, 2, 3, 6, 9
    // Example float array:        1.25, 3.50, 6.75
    // Example complexarray:       "(1.25, 2.5) (2.4, -2.3) (-1.2, -2.2)"
    // Example stringarray:        "Button 1" "Button 2"
units {UNITLESS_UNIT}
    // Options: STRING_UNIT, UNITLESS_UNIT,
    // FREQUENCY_UNIT, TIME_UNIT, ANGLE_UNIT
    // Note: ANGLE_UNIT for phase in degrees
    // Other units are available for resistance, length, etc., but might
    // not be relevant to numeric stars
desc {my_state_description}
    // Begin with a short phrase defining this state and ending with
    // a period and a space. This initial sentence will be used in the
    // AEL for this star. This initial sentence may be followed
    // with additional content to describe this state and its use
attributes {A_SETTABLE | A_NONCONSTANT}
    // These attributes are for states the for use at the schematic
    // level. If a state is to be hidden from the schematic, it can be
    // listed as A_NONSETTABLE | A_NONCONSTANT
}
// Define a defstate for each parameter; example for enumerated state
defstate {
    name {my_state_name} // Limit length to one line 30 characters maximum
                        // No spaces. Only alpha-numeric characters and underbar.
                        // Name should be constructed as one or more word segments
                        // with each word segment beginning with a capital letter.
    type {enum} // enumerated state
    default {"option1"} // default in quotes
    desc {my_state_description}
        // Same notes as for the state description apply
    enumlist { option 1, option 2 }
        // enumerated list separate with commas
        // each enumeration may contain spaces, underbar or other
        // alpha-numeric characters, but none other
        // Code may be reference the enumeration by use of the option
        // with spaces replaced by underbars.
        // Example: if (my_enum_name == option_1) {
        //     ... code here ...
        // }
    enumlabels { opt 1, opt 2 }
        // an abbreviation of the enumlist options, used during AEL
        // generation
    attributes {A_SETTABLE|A_NONCONSTANT}
        // These attributes are for states the for use at the schematic
        // level. If a state is to be hidden from the schematic, it can be
        // listed as A_NONSETTABLE | A_NONCONSTANT
}
// Define a defstate for each parameter; example for file name state
defstate {
    name {my_state_name} // Limit length to one line 30 characters maximum
                        // No spaces. Only alpha-numeric characters and underbar.
                        // Name should be constructed as one or more word segments
                        // with each word segment beginning with a capital letter.
    type {filename} // file name state
    default {"xyz.ext1"} // default in quotes
    desc {my_state_description}
        // Same notes as for the state description apply
    extensions { ext1, ext2, ext3 }
        // extension list separate with commas
        // each extension may contain underbar or other
        // alpha-numeric characters, but none other
    attributes {A_SETTABLE|A_NONCONSTANT}
        // These attributes are for states the for use at the schematic
        // level. If a state is to be hidden from the schematic, it can be
        // listed as A_NONSETTABLE | A_NONCONSTANT
}
port_type { // Options: input, output, inmulti, outmulti

```

User-Defined Models

```

        // See programmers documentation for the use of each port_type
        name {port_name}
        type {port_type}
// When Domain == SDF:
//Options: int, float, fix, complex, message, int_matrix_env,
        //      float_matrix_env, complex_matrix_env, fix_matrix_env,
        //      anytype
// When Domain == TSDF:
// Options: timed
        desc {port_desc}
}
hinclude {
// Optional: delete if not used
// User specifies other files to include in the .h file
}
header {
// Optional: delete if not used
// User places C/C++ code to include in the .h file, before the class definition
}
ccinclude {
// Optional: delete if not used
// User inserts .cc include files here in quotes with comma separators. Example:
// "file1.cc","file2.cc","file3.cc"
}
private {
// Optional: delete if not used
// Define private data members of the star class
}
protected {
// Optional: delete if not used
// Define protected data members of the star class
}
constructor {
// Optional: delete if not used
// Called when instance created.
// Allows user to specify extra C/C++ code to be executed in the constructor
// for the class.
// This field can initialize the public data member that indicates delays associated
// with input pins
}
conscalls {
// Optional: delete if not used
// Used when data members have constructors and require arguments.
// These members would be added by using the public, private, or
// protected keywords. If such members exist, conscalls provides
// the user with a mechanism for passing arguments to the
// constructors of those members. Example:
// member1(arglist), member2(arglist)
}
setup {
// Optional: delete if not used
// C/C++ code to execute at start time, before the scheduler setup.
// Check each state value for validity
// Example: if (double(state_name) < 0.5) {
//             state_name.rangeError(">= 0.5");
//         }
// See also messaging guidelines for status messages, warning messages,
// error messages
}
begin {
// Optional: delete if not used
// C/C++ code to execute at start time, after the scheduler setup.
}
go {
// User supplied C/C++ code here
//
}
wrapup {
// Optional: delete if not used
// C/C++ code to invoke at the end of a run (if no error occurred)
}
destructor {
// Optional: delete if not used
// User C/C++ code to include in the destructor for the star
}
}

```

```

method {
// Optional: delete if not used
// Define a member function for the star class
// Can also substitute for method:
// virtual method, inline method, pure method, pure virtual method,
// inline virtual method
// name {user defined name}
// access {either private, protected, or public}
// arglist {"(arguments in quotes)"}
// type {the return type of the method}
// code {C/C++ code defining the method}
}
code {
// Optional: delete if not used
// C/C++ code to include in the .cc file outside the class definition
}
}

```

Writing Sink Models

Sinks are models with inputs but no outputs. The main use of sinks is to write data to files (ASCII, dataset, etc.). The data written to the file could be the raw data collected from the sink's inputs, or the sink's collected data can be processed and then written to the file. The processing of the data can be done during the simulation (in the go method of the sink) or after the simulation has finished (in the wrapup method of the sink).

In order to write a sink model, you first need to understand the concept of a task. In addition, you need to learn how to use tasks and how to write data to a dataset. The following sections describe these concepts and give simple examples showing how they are used.

Understanding Tasks

A task is something that needs to be completed before the simulation can finish. In other words, you can think of a task as something that controls the simulation by keeping it running or by causing it to terminate. The simulator keeps a list of all the tasks that have not completed, called the TaskList, and as long as there are tasks in this list it will continue to run the simulation. The simulation terminates as soon as the TaskList becomes empty.

Any model can add/remove tasks to/from the TaskList at any time during the simulation. However, tasks are particularly useful when used in source or sink models. All ADS Ptolemy sinks and a few sources (the ones that generate a finite amount of data, such as the file-based ones) use tasks to control how long the simulation will run. Some of the Interactive Control and Displays components also use tasks.

For example, a file-based source adds a task to the TaskList at the beginning of the simulation and removes it when it has reached the end of the file it reads. This way (and if no other component has added a task to the TaskList) you can guarantee that the simulation will run as long as there is data in the file being read.

On the other hand, sinks typically add a task to the TaskList at the beginning of the

simulation and remove it when they have collected all the data they need. The time a sink removes its task from the TaskList is usually known before the simulation starts (almost all sinks have a Stop parameter). However, there are sinks that decide when to remove their tasks while the simulation is running. Examples of such sinks are the BER sinks (berIS, berMC, berMC4), which keep track of the relative variance of their BER estimate and remove their tasks when the variance falls below a user-specified value. The SimpleBERSink shown in the section *SDFSsimpleBERSink* is another example of such a sink.

Sink Coding Methodology

Every sink needs to define an object of the class SinkControl in the private, protected, or public section of its .pl file. This will require that the TargetTask.h header file is listed in the hinclude section of the .pl file and that the variable KERNEL (or another variable that automatically sets KERNEL to 1, such as SDFKERNEL, TSDFKERNEL) is set to 1 in the corresponding make-defs file. The SinkControl object should call its initialize function in the begin section. The go section should look like:

```
go {
if (sinkControl.collectData()) {
// all the sink go code should be entered here.
}
}
```

where sinkControl is an object of type SinkControl.

Let's look into what all the above means in more detail. A SinkControl object is an object that can add/remove a single task to/from the TaskList. In addition, it has an internal timer/counter to keep track of some notion of the simulation time. A SinkControl object needs to be initialized before it can be used. This is done by calling its initialize function. There are two overloaded versions of this function:

i) initialize(Block& master, double start_value, double stop_value, double step_value)

ii) initialize(Block& master, double start_value, double step_value)

The first argument in both cases must always be *this, where this is the pointer to the sink object itself. The choice of which initialize function is called will determine the way the sink behaves:

1. When the first initialize function is called, the SinkControl object will add a task to the TaskList and reset its internal timer/counter to 0. Then every time the collectData function is called, it increments the timer/counter by step_value and returns 1 (TRUE) if the timer/counter (before being incremented) was between start_value and stop_value. It returns 0 (FALSE) otherwise. When the timer/counter reaches stop_value the task is automatically removed from the TaskList. If you want to remove the task from the TaskList before stop_value is reached, you can do so by calling the stopControl function. This is used in the BER sinks (see example SimpleBERSink at the end of this section) where stopControl is called when some

condition is satisfied. Although in this case you might want to ignore `stop_value` completely, it still makes sense to define it as an upper limit of how long the simulation will run just in case the condition that needs to be satisfied before `stopControl` is called is never satisfied. If you are absolutely certain that the condition you are using will be satisfied, and do not know what value to use for `stop_value`, use a very large value, e.g., `1.0e20`. When calling this initialize function, `start_value` must be greater than or equal to 0, `stop_value` must be greater than or equal to `start_value`, and `step_value` must be greater than 0. If these conditions are not satisfied the simulation will abort.

2. When the second initialize function is called, the `SinkControl` object will not add a task to the `TaskList`. Therefore, the sink will not control how long the simulation will run. The timer/counter is still reset to 0. When `collectData` is called, it increments the timer/counter by `step_value` and returns 1 (TRUE) if the timer/counter (before being incremented) was greater than or equal to `start_value`. In this mode of operation, the sink can be used to collect all the data from a simulation controlled by some other sink or source. This may be useful for large, multirate designs, where you do not know the rates at the points where you want to collect the data. If the sink's `SinkControl` object is initialized using the first initialize function and `start_value`, `stop_value` are not selected appropriately, the design might end up simulating a lot more than it should. By setting only one sink to control the simulation and letting the others collect data as long as the simulation runs, you can guarantee that the data collected in all sinks will correspond time-wise to the data collected in the sink that controlled the simulation. Another case where this mode of operation is useful is when the input signal for a simulation is read from a file and the amount of data in the file is not known. By setting only the source to control the simulation, you can "force" the sinks to collect the right amount of data no matter how much data there is in the file. When calling this initialize function, `start_value` must be greater than or equal to 0 and `step_value` must be greater than 0. If these conditions are not satisfied, the simulation will abort.

Useful Notes/Hints

- A sink model need not operate in only one of the two ways described above. Parameters can be used to decide how the sink's `SinkControl` object is initialized. For example, this is the purpose of the `ControlSimulation` parameter of the `NumericSink` and `TimedSink` models. Also see the examples in the section [Examples of Sink Models](#).
- When a sink is to be used with numeric data, it is recommended to use 1 as the `step_value` when calling the initialize function. When a sink is to be used with timed data, it is recommended to use the simulation time step (obtained by calling `input.getTimeStep()`, where `input` is the name of the input port) as the `step_value` when calling the initialize function.
- It is recommended that you only write uni-rate sink models, that is, sinks that only read one sample from their inputs every time they are fired.
- A sink that needs to post-process the data it collects, that is, it just stores the data in some array during `go` and processes it in `wrapup`, must always be initialized using the first initialize function. Otherwise, you will not know how much memory needs to be allocated for the array that will store the collected data during `go`. Examples of sinks like that are the `SpectrumAnalyzer`, `EVM`, and refer to `SDFMedianSink`. A sink that

can process the collected data in go can be initialized in either of the two ways described above.

- Two other useful functions of the *SinkControl* class are the *time* and *index* functions, which return the current value of the *SinkControl* object's internal timer/counter. The *time* function returns a double and it should be used with timed data, whereas the *index* function returns an int and it should be used with numeric data. These functions are typically used as the value of the independent variable for data written to a dataset.
- The "Data collection is XX.X% complete" messages displayed in the Status/Summary window are automatic (there is nothing extra you need to do in order to get these messages printed out). However, if you want to print more status information you can use the `Error::warn()` or `Error::message()` methods. The `Error::warn()` method sends messages to the Simulation/Synthesis messages window, whereas the `Error::message()` method sends messages to the Status/Summary window. For more details on these methods, refer to [Messaging Guidelines for Star .pl Files](#) in this section. Refer to *SDFSsimpleBERSink* for an example of how the `Error::message()` method can be used.

Writing Data to a Dataset

If the sink model needs to write data to a dataset, it needs to make use of the *SimData* class. Not all sinks write data to a dataset. For example, the Printer sink writes data to an ASCII file. To use the *SimData* class you need to define a pointer to an object of this class in the private, protected, or public section of the sink's .pl file. This will require that the *SimData.h* header file is listed in the *ccinclude* section of the .pl file and that the variable *KERNEL* (or another variable that automatically sets *KERNEL* to 1, such as *SDFKERNEL*, *TSDFKERNEL*) is set to 1 in the corresponding make-defs file.

The *SimData* class is an abstract class so only pointers to it can be defined. If you define an object of this class in your sink model, your model will not even compile. The compiler will error out with an error message similar to the ones below:

- `SimData : cannot instantiate abstract class (Windows)`
- `Cannot declare a member of the abstract type SimData (Sun)`
- `A class member may not be declared with an abstract class type (HP-UX)`

What follows describes the use of the functions of the *SimData* class. While reading the following paragraphs keep in mind that data written to a dataset always has a dependent and independent variable associated with it.

- *newSimData* (Block **starP*). This is not a function of the *SimData* class but it must be called in order to initialize the pointer to the *SimData* class. This must be done before the pointer can be used. The argument of this function must always be *this*, where *this* is the pointer to the sink object itself. For example, if you have defined a pointer to an object of type *SimData* and its name is *dataP*, then the following piece of code should precede any use of *dataP*:

```
dataP = newSimData(this);
```
- `setIndepVar(const char *name, AgilentPtolemy:: DataType type, State::Unit unit)` is

- used to set a name, type and unit for the independent variable of the data. The value of type can be `AgilentPtolemy::INT` (the independent variable will be of integer type; typically used for numeric data) or `AgilentPtolemy::FLOAT` (the independent variable will be of double type; typically used for timed data). The value of unit can be `State::UNITLESS_UNIT` (the independent variable will have no associated unit; typically used for numeric data), `State::TIME_UNIT` (the independent variable will represent time; typically used for timed data), or `State::FREQUENCY_UNIT` (the independent variable will represent frequency; typically used for spectrum data).
- `setDepVar(const char *name, AgilentPtolemy:: DataType type, State::Unit unit)` is used to set a name, type and unit for the dependent variable of the data. The value of type can be `AgilentPtolemy::INT` (the dependent variable will be of integer type), `AgilentPtolemy::FLOAT` (the dependent variable will be of double type), or `AgilentPtolemy::COMPLEX` (the dependent variable will be of complex type). The `AgilentPtolemy::FIX` data type, as well as all the `AgilentPtolemy::MATRIX` data types, are not supported. The value of unit can be `State::UNITLESS_UNIT` (the dependent variable will have no associated unit; typically used for numeric data), `State::VOLTAGE_UNIT` (the dependent variable will represent voltage; typically used for timed data), or `State:: POWER_UNIT` (the dependent variable will represent power in dBm; typically used for spectrum data). A unique name must be selected for the name of the dependent variable. A way to obtain a unique name is to call the `fullName()` function which returns the instance name of the sink, for example `N1` or `X1.N1` (if the sink is inside a subnetwork with instance name `X1`).
 - `setDepVar(const char *baseName, const char *suffix, AgilentPtolemy:: DataType type, State::Unit unit)` is an overloaded version of the `setDepVar` function that can be used to give the dependent variable the name `baseName.suffix`. This is useful when a sink writes multiple variables to the dataset.
 - `setAutoPlotType(const int &type)` can be used to automatically plot data at the end of the simulation. If the value of 0 is passed, no automatic plotting occurs. If the value of 1 is passed, a rectangular plot is automatically plotted at the end of the simulation.
 - `addAttribute(const char *name, int value)` and `addAttribute(const char *name, double value)` can be used to associate integer or double attributes with the data. For example, the `TimedSink` uses this function to associate a characterization frequency with its data. To retrieve the value of an attribute in the Data Display window, you have to use the function `get_attr(sinkName, attributeName)`, e.g., `char_freq = get_attr(T1, "fc")`.
 - `sendData(x, y)` (six overloaded versions) is used to send data to the dataset.

Typically, a sink handles only one type of data. However, there is no such limitation. A sink can be written to handle any type of data by declaring its input to be of type `ANYTYPE`. To get the type of data that the sink has received in a particular simulation, the method `PortHole::resolvedType()` can be called. Then according to the value this method returns, you can call the `setDepVar()` and `sendData()` methods with the appropriate arguments to handle the specific data type the sink has received. Refer to the example *SDFSimpleNumericSink*, which can handle `AgilentPtolemy::INT`, `AgilentPtolemy::FLOAT`, and `AgilentPtolemy::COMPLEX` data.

Examples of Sink Models

This section lists five examples of sink models:

- SDFSsimpleNumericSink.html
- TSDFSsimpleTimedSink.html
- SDFSsimpleBERSink.html
- SDFMeanVarianceSink.html
- SDFMedianSink.html

These examples can be found in the directory *doc/sp_items* under your ADS installation directory. Open each file to see the complete C code for that example.

The source code for these sinks can be found in the directory *doc/sp_items* under your ADS installation directory. The names of the source files are:

- SDFSsimpleNumericSink.pl
- TSDFSsimpleTimedSink.pl
- SDFSsimpleBERSink.pl
- SDFMeanVarianceSink.pl
- SDFMedianSink.pl.

The make-defs file used to compile these models is also found in the same directory.

The first two examples, SimpleNumericSink and SimpleTimedSink, are sinks that just write the raw data they collect to the dataset. The SimpleBERSink is a sink that processes the collected data in its go method and writes data in the go as well as the wrapup method. The MeanVarianceSink processes the data in go and writes the results in wrapup. The MeadianSink is an example of a post-processing sink; it just collects the data in go and does all the processing as well as writes the results to the dataset in the wrapup method.

Data Types for Model Builders

Stars communicate by sending objects of type *Particle*. A basic set of types, including scalar and array types, built on the Particle class, is built into the ADS Ptolemy kernel. Since all of these particle types are derived from the same base class, it is possible to write stars that operate on any of them (by referring only to the base class). It is also possible to define new types that contain arbitrary C++ objects.

There are currently eleven key data particle types defined in the ADS Ptolemy kernel. There are four numeric scalar types-complex, fixed-point, double precision floating-point, and integer-described in the section [Scalar Numeric Types](#). The fixed-point scalar type has two forms, UCB (University of California at Berkeley) fixed-point and Agilent fixed-point. The Agilent fixed-point is a superset of the UCB Ptolemy fixed-point.

ADS Ptolemy supports user-defined types-using the class Message, described in the section [Defining New Data Types](#). Each of the scalar numeric types has an equivalent matrix type, which uses a more complex version of the user-defined type mechanism; these are described in the section [The Matrix Data Types](#).

With ADS Ptolemy, you may write stars that will read and write particles of any type; this mechanism is described in the section [Writing Stars That Manipulate Any Particle Type](#). There is also the timed signal type with two forms, Baseband and Complex Envelope, described in the section, [Timed Particle Signal Type](#).

Scalar Numeric Types

There are four scalar numeric data types defined in the ADS Ptolemy kernel: complex, fixed-point, double precision floating-point, and integer. All of these can be read from and written to portholes as described in the section *Reading Inputs and Writing Outputs* (modbuild). The floating-point and integer data types are based on the standard C++ *double* and *int* types, and need no further explanation. To support the other two types, the ADS Ptolemy kernel contains a *Complex* class and a *Fix* class, which are described in the remainder of this section.

The Complex Data Type

The Complex data type in ADS Ptolemy contains real and imaginary components, each of which is specified as a double precision floating-point number. The notation used to represent a complex number is a two-number pair: (*real*, *imaginary*). For example, (1.3,-4.5) corresponds to the complex number $1.3 - 4.5j$. Complex implements a subset of the functionality of the complex number classes in the cfront and libg++ libraries, including most of the standard arithmetic operators and a few transcendental functions.

Constructors

Complex()

Create a complex number initialized to zero-that is, (0.0, 0.0). For example, Complex C.

Complex(double real, double imag)

Create a complex number whose value is (*real*, *imaginary*). For example, Complex C(1.3,-4.5).

Complex(const Complex& arg)

Create a complex number with the same value as the argument (the copy constructor). For example, Complex A(complexSourceNumber).

Basic Operators

The following list of arithmetic operators modify the value of the complex number. All functions return a reference to the modified complex number (**this*).

```
Complex& operator = (const Complex& arg)
Complex& operator += (const Complex& arg)
Complex& operator -= (const Complex& arg)
Complex& operator *= (const Complex& arg)
Complex& operator /= (const Complex& arg)
Complex& operator *= (double arg)
Complex& operator /= (double arg)
```

There are two operators to return the real and imaginary parts of the complex number:

```
double real() const
double imag() const
```

Non-Member Functions and Operators

The following one- and two-argument operators return a new complex number:

```
Complex operator + (const Complex& x, const Complex& y)
Complex operator - (const Complex& x, const Complex& y)
Complex operator * (const Complex& x, const Complex& y)
Complex operator * (double x, const Complex& y)
Complex operator * (const Complex& x, double y)
Complex operator / (const Complex& x, const Complex& y)
Complex operator / (const Complex& x, double y)
Complex operator - (const Complex& x)
```

Return the negative of the complex number.

```
Complex conj (const Complex& x)
```

Return the complex conjugate of the number.

```
Complex sin(const Complex& x)
Complex cos(const Complex& x)
Complex exp(const Complex& x)
Complex log(const Complex& x)
Complex sqrt(const Complex& x)
Complex pow(double base, const Complex& expon)
Complex pow(const Complex& base, const Complex& expon)
```

Other general operators:

```
double abs(const Complex& x)
```

Return the absolute value, defined to be the square root of the norm.

```
double arg(const Complex& x)
```

Return the value $\arctan(x.\text{imag}()/x.\text{real}())$.

```
double norm(const Complex& x)
```

Return the value $x.\text{real}()^2 + x.\text{imag}()^2$.

```
double real(const Complex& x)
```

Return the real part of the complex number.

```
double imag(const Complex& x)
```

Return the imaginary part of the complex number.

Comparison Operators:

```
int operator != (const Complex& x, const Complex& y)
int operator == (const Complex& x, const Complex& y)
```

The Fixed-Point Data Type

The fixed-point data type is implemented in ADS Ptolemy by the `Fix` class. The former supports a two's complement representation of a finite precision number. The latter supports a two's complement and an unsigned representation of a finite precision number. In fixed-point notation, the partition between the integer part and the fractional part, the binary point, lies at a fixed position in the bit pattern. Its position represents a trade-off between precision and range. If the binary point lies to the right of all bits, then there is

no fractional part.

The fixed-point number has a form specified by arithmetic type (ArithType: 2's complement or unsigned), bitwidth, and number of fractional bits. The bitwidth and number of fractional bits compose the precision of the fixed-point number. The precision is specifiable with either of these two forms:

$x.y$ or y/n

where

x = number of integer bits (including sign bit) to the left of the decimal point

y = number of fractional bits to the right of the decimal point

n = total number of bits (bitwidth)

Fixed-point operations include consideration of overflow type (wrapped, saturate, saturate-to-zero) and quantization type (round or truncate).

ADS Ptolemy Fix class is an extension of the UCB Fix class. The distinction between the UCB and Agilent fixed-point data types is as follows. Note that a distinction is made between Synthesizable DSP components (such as those found in the Numeric Synthesizable library) and those that are not.

Fix Data Type Properties

Attributes for Fix Data Type	UCB Fixed-Point Stars	Agilent Fixed-Point Stars
ArithType, default	2's complement	2's complement
ArithType, options.	2's complement	Synthesizable DSP-2's complement, unsigned Non-synthesizable DSP-2's complement
Precision, max bit width	32	256
Overflow handler, default	saturate	Synthesizable DSP-wrapped Non-synthesizable DSP- saturate
Overflow handler, options	wrapped, saturate, saturate-to-zero	Synthesizable DSP-wrapped, saturate Non-synthesizable DSP-wrapped, saturate, saturate-to-zero
RoundFix, default	truncate	truncate
RoundFix, options.	round, truncate	round, truncate
Generates Verilog or VHDL	No	Synthesizable DSP-Yes Non-synthesizable DSP-No

Constructing Fixed-Point Variables

Variables of type Fix are defined by specifying the word length and the position of the binary point. At the user-interface level, precision is specified either by setting a fixed-point parameter to a (*value, precision*) pair, or by setting a *precision* parameter. The former gives the value and precision of some fixed-point value, while the latter is typically used to specify the internal precision of star computations. In either case, the syntax of the precision is either $x.y$ or m/n , where x is the number of integer bits (including the sign bit), y and m are the number of fractional bits, and n is the total number of bits. Thus, the total number of bits in the fixed-point number (also called its *length*) is $x + y$ or n . For

example, a fixed-point number with precision 3.5 has a total length of 8 bits, with 3 bits to the left and 5 bits to the right of the binary point.

At the source code level, methods working on Fix objects either have the precision passed as an *x.y* or *m/n* string, or as two C++ integers that specify the total number of bits and the number of integer bits including the sign bit (that is, *n* and *x*). For example, suppose you have a star with a precision parameter named *precision*. Consider the following code:

```
Fix x = Fix(((const char *) precision));
if (x.invalid())
    Error::abortRun(*this, "Invalid precision");
```

The *precision* parameter is cast to a string and passed as a constructor argument to the Fix class. The error check verifies that the precision was valid.

There is a maximum value for the total length of a Fix object which is 256 bits. Numbers in the Fix class represented using two's complement notation have the sign bit stored in the bits to the left of the binary point. There must always be at least one bit to the left of the binary point to store the sign for two's complement arithmetic type.

In addition to its value, each Fix object contains information about its precision and error codes indicating overflow, divide-by-zero, or bad format parameters. The error codes are set when errors occur in constructors or arithmetic operators. There are also fields to specify:

- Whether rounding or truncation takes place when other Fix values are assigned to it-truncation is the default;
- The response to an overflow or underflow on assignment-the default is saturation for UCB fixed-point, and wrapped for Agilent fixed-point (see [Assignment and Overflow Handling](#)).

Fixed-Point States

State variables can be declared as either *Fix* or *FixArray*. The precision is determined by an associated precision state using either of two syntaxes:

- Specifying only a value in the dialog box creates a fixed-point number with the default length of 32 bits, and with the position of the binary point set as required to store the integer value. For example, the value *1.0* creates a fixed-point object with precision 2.30, and the value *0.5* creates one with precision 1.31.
- Specifying both a value and a precision creates a fixed-point number with the stipulated precision. For example, for "(*value*, *precision*)" = "(2.546, 3.5)", a fixed-point object is created by casting the double 2.546 to a Fix with precision 3.5. Note that it is mandatory to use parenthesis when specifying "(*value*, *precision*)" in the dialog box.

Fixed-Point Inputs and Outputs

Fix types are available in ADS Ptolemy as a type of particle. The automatic conversion from an *int* or a *double* to a *Fix* takes place using the `Fix::Fix(double)` constructor, which makes a *Fix* object with the default word length of 32 bits and the number of integer bits as required by the value. For instance, the *double* 10.3 will be converted to a *Fix* with precision 5.19, since 5 is the minimum number of bits needed to represent the integer part, 10, including its sign bit. However, there is no automatic conversion to the ADS Ptolemy *Fix* type for use with synthesizable DSP components. The user must explicitly cast an *int* or *double* particle to a synthesizable *fix* particle using a Signal Converter (*FloatToFixSyn*).

To use the *Fix* type in a star, the type of the portholes must be declared as *fix*.

Stars that receive or transmit fixed-point data have parameters that specify the precision of the input and output in bits, as well as the overflow behavior. Here is a simplified version of the `SDFAddFix` star, configured for two inputs:

```
defstar {
    name {AddFix}
    domain {SDF}
    derivedFrom{SDFFix}
    input {
        name {input1}
        type {fix}
    }
    input {
        name {input2}
        type {fix}
    }
    output {
        name {output}
        type {fix}
    }
    defstate {
        name {OutputPrecision}
        type {precision}
        default {2.14}
    }
    desc {
        Precision of the output in bits and precision of the accumulation.
        When the value of the accumulation extends outside of the precision,
        the OverflowHandler will be called.
    }
}
```

Note that the real `AddFix` star supports any number of inputs. By default, the precision used by this star during the addition will have 2 bits to the left of the binary point and 14 bits to the right. Not shown here is the state `OverflowHandler`, inherited from the `SDFFix` star, which defaults to *saturate* -that is, if the addition overflows, then the result saturates, pegging it to either the largest positive or negative number representable. The result value, *sum*, is initialized by the following code:

```
protected {
    Fix sum;
}
begin {
    SDFFix::begin();
    sum = Fix(((const char *) OutputPrecision));
    if (sum.invalid())
        Error::abortRun(*this, "Invalid OutputPrecision");
    sum.set_ovflow(((const char*) OverflowHandler.enumString ((int) OverflowHandler)));
}
```

```

    if (sum.invalid())
        Error::abortRun(*this, "Invalid OverflowHandler");
}

```

The *begin* method checks the specified precision and overflow handler for correctness. Then, in the *go* method, we use *sum* to calculate the result value, thus guaranteeing that the desired precision and overflow handling are enforced. For example,

```

go {
    sum.setToZero();
    sum += Fix(input1%0);
    checkOverflow(sum);
    sum += Fix(input2%0);
    checkOverflow(sum);
    output%0 << sum;
}

```

(The `checkOverflow` method is inherited from `SDFFix`.) The protected member *sum* is an uninitialized `Fix` object until the *begin* method runs. In the *begin* method, it is given the precision specified by `OutputPrecision`. The *go* method initializes it to zero. If the *go* method had instead assigned it a value specified by another `Fix` object, then it would acquire the precision of that other object—at that point, it would be initialized.

Assignment and Overflow Handling

Once a `Fix` object has been initialized, its precision does not change as long as the object exists. The assignment operator is overloaded so that it checks whether the value of the object to the right of the assignment fits into the precision of the left object. If not, then it takes the appropriate overflow response and sets the overflow error bit.

If a `Fix` object is created using the constructor that takes no arguments, as in the *protected* declaration above, then that object is an uninitialized `Fix`; it can accept any assignment, acquiring not only its value, but also its precision and overflow handler.

The behavior of a `Fix` object on an overflow depends on the specifications and the behavior of the object itself. Each object has a private data field that is initialized by the constructor; when there is an overflow, the `overflow_handler` looks at this field and uses the specified method to handle the overflow. This data field is set to *saturate* by default, and can be set explicitly to any other desired overflow handling method using a function called `set_ovflow(<keyword>)`. The keywords for overflow handling methods are: *saturate* (default), *zero_saturate*, *wrapped*, and *warning*. With *saturate*, the original value is replaced by the maximum (for overflow) or minimum (for underflow) value representable given the precision of the `Fix` object. *zero_saturate* sets the value to zero.

Explicitly Casting Inputs

In the above example, the first line of the *go* method assigned the input to the protected member *sum*, which has the side-effect of quantizing the input to the precision of *sum*. Alternatively, we could have written the *go* method as follows:

```

go {
  sum = Fix(input1%0) + Fix(input2%0);
  output%0 << sum;
}

```

The behavior here is significantly different: the inputs are added using their own native precision, and only the result is quantized to the precision of *sum*.

Some stars enable you to select between these two different behaviors with a parameter called *UseArrivingPrecision*. If set to *YES*, the input particles are not explicitly cast; they are used as they are; if set to *NO*, the input particles are cast to an internal precision, which is usually specified by another parameter.

Here is the (abbreviated) source of the SDFGainFix star, which demonstrates this point:

```

defstar {
  name {GainFix}
  domain \{SDF\}
  derivedFrom {SDFFix}
  desc {
    This is an amplifier; the fixed-point output is the fixed-point input
    multiplied by the "gain" \{(default 1.0)\}. The precision of "gain", the
    input, and the output can be specified in bits.
  }
  input {
    name {input}
    type {fix}
  }
  output {
    name {output}
    type {fix}
  }
  defstate {
    name {gain}
    type {fix}
    default {1.0}
    desc {Gain of the star.}
  }
  defstate {
    name {UseArrivingPrecision}
    type {int}
    default {"YES"}
    desc {
      Flag indicating whether or no to use the arriving particles
      as they are: YES keeps the same precision, and NO casts them
      to the precision specified by the parameter "InputPrecision".
    }
  }
  defstate {
    name {InputPrecision}
    type {precision}
    default {2.14}
    desc {
      Precision of the input in bits. The input particles are only cast
      to this precision if the parameter "ArrivingPrecision" is set to NO.
    }
  }
  defstate {
    name {OutputPrecision}
    type {precision}
    default {2.14}
    desc {
      Precision of the output in bits. This is the precision that will
      hold the result of the arithmetic operation on the inputs. When
      the value of the product extends outside of the precision, the
      OverflowHandler will be called.
    }
  }
}

```



```

protected {
    Fix fixIn, out;
}
begin {
    SDDFix::begin();
    if (! int(UseArrivingPrecision)) {
        fixIn = Fix(((const char *) InputPrecision));
        if(fixIn.invalid())
            Error::abortRun(*this, "Invalid InputPrecision");
    }
    out = Fix(((const char *) OutputPrecision));
    if (out.invalid())
        Error::abortRun(*this, "Invalid OutputPrecision");
    out.set_ovflow(((const char*) OverflowHandler.enumString
        ((int) OverflowHandler)));
    if(out.invalid())
        Error::abortRun(*this, "Invalid OverflowHandler");
}
go {
    // all computations should be performed with out since
    // that is the Fix variable with the desired overflow
    // handler
    out = Fix(gain);
    if (int(UseArrivingPrecision)) {
        out *= Fix(input%0);
    }
    else {
        fixIn = Fix(input%0);
        out *= fixIn;
    }
    checkOverflow(out);
    output%0 << out;
}
// a wrap-up method is inherited from SDDFix
// if you defined your own, you should call SDDFix::wrapup()
}

```

Note that *SDFGainFix* star, like many other Fix stars, is derived from the star *SDFFix*. *SDFFix* implements commonly used methods and defines two states: *OverflowHandler* selects one of four overflow handlers to be called each time an overflow occurs; and *ReportOverflow*, if TRUE, causes the number and percentage of overflows that occurred for that star during a simulation run to be reported in the *wrapup* method.

Constructors

Fix()

Create a Fix number with unspecified precision and value zero.

Fix(int length, int intbits)

Create a Fix number to the left of the binary point with total word length of length bits and intbits bits. The value is set to zero. If the precision parameters are not valid, an error bit is internally set so that the invalid method returns TRUE.

Fix(const char* precisionString)

Create a Fix number whose precision is determined by precisionString, with the syntax leftbits.rightbits, where leftbits is the number of bits to the left of the

binary point and rightbits is the number of bits to the right of the binary point; or rightbits/totalbits, where totalbits is the total number of bits. The value is set to zero. If precisionString is not in the proper format, an error bit is internally set so that the invalid method will return TRUE.

`Fix(double value)`

Create a Fix with the default precision of 24 total bits for the word length and set the number of integer bits to the minimum needed to represent the integer part of the number value. If the value given needs more than 24 bits to represent, the value will be clipped and the number stored will be the largest possible under the default precision (that is, saturation occurs). In this case, an internal error bit is set so that the `ovf_occurred` method will return TRUE.

`Fix(int length, int intbits, double value)`

Create a Fix with the specified precision and set its value to the given value. The number is rounded to the closest representable number given the precision. If the precision parameters are not valid, then an error bit is internally set so that the invalid method will return TRUE.

`Fix(const char* precisionString, double value)`

Same as the previous constructor except that the precision is specified by the given precisionString instead of as two integer arguments. If the precision parameters are not valid, then an error bit is internally set so that the `invalid()` method will return TRUE when called on the object.

`Fix(const char* precisionString, uint16* bits)`

Create a Fix with the specified precision and set the bits precisely to the ones in the given bits. The first word pointed to by bits contains the most significant 16 bits of the representation. Only as many words as are necessary to fetch the bits will be referenced from the bits argument. For example: `Fix("2.14",bits)` will only reference `bits[0]`.

`Fix(const Fix& arg)`

Copy constructor. Produces an exact duplicate of arg.

`Fix(int length, int intbits, const Fix& arg)`

Read the value from the Fix argument and set to a new precision. If the precision parameters are not valid, then an error bit is internally set so that the invalid method will return TRUE when called on the object. If the value from the source will not fit, an error bit is set so that the `ovf_occurred` method will return TRUE.

Functions to Set Or Display Information about the Fix Number

`int len() const`

Returns total word length of the Fix number.

`int intb() const`

Returns number of bits to the left of the binary point.

`int precision() const`

Returns number of bits to the right of the binary point.

`int overflow() const`

Returns the code of the overflow response type for the Fix number. The possible codes are:

0 = `ovf_saturate`

1 = `ovf_zero_saturate`

2 = `ovf_wrapped`

3 = `ovf_warning`

4 = `ovf_n_types`

`int roundMode() const`

Returns the rounding mode: 1 for rounding, 0 for truncation.

`int signBit() const`

Returns TRUE if the value of the Fix number is negative, FALSE if it is positive or zero.

`int is_zero()`

Returns TRUE if the value of Fix is zero.

`double max()`

Returns the maximum value representable using the current precision.

`double min()`

Returns the minimum value representable using the current precision.

`double value ()`

The value of the Fix number as a double.

`void setToZero ()`

Sets the value of the Fix number to zero.

```
void set_overflow (int value)
```

Sets the overflow type.

```
void set_rounding (int value)
```

Sets the rounding type: TRUE for rounding, FALSE for truncation.

```
void initialize()
```

Discards the current precision format and set the Fix number to zero.

There are a few functions for backward compatibility:

```
void set_ovflow(const char*)
```

Sets the overflow using a name.

```
void Set_MASK(int value)
```

Sets the rounding type. Same functionality as set_rounding().

Comparison function:

```
int compare (const Fix& a, const Fix& b)
```

Compares two Fix numbers. Return -1 if $a < b$, 0 if $a = b$, 1 if $a > b$.

The following functions are for use with the error condition fields:

```
int ovf_occurred ()
```

Returns TRUE if an overflow has occurred as the result of some operation like addition or assignment.

```
int invalid ()
```

Returns TRUE if the current value of the Fix number is invalid due to it having an improper precision format, or if some operation caused a divide by zero.

```
int dbz ()
```

Returns TRUE if a divide by zero error occurred.

```
void clear_errors ()
```

Resets all error bit fields to zero.

Operators

`Fix& operator = (const Fix& arg)`

Assignment operator. If `*this` does not have its precision format set (that is, it is uninitialized), the source `Fix` is copied. Otherwise, the source `Fix` value is converted to the existing precision. Either truncation or rounding takes place, based on the value of the rounding bit of the current object. Overflow results either in saturation, "zero saturation" (replacing the result with zero), or a warning error message, depending on the overflow field of the object. In these cases, `ovf_occurred` will return `TRUE` on the result.

`Fix& operator = (double arg)`

Assignment operator. The double value is first converted to a default precision `Fix` number and then assigned to `*this`.

The function of these arithmetic operators should be self-explanatory:

```
Fix& operator += (const Fix&)
Fix& operator -= (const Fix&)
Fix& operator *= (const Fix&)
Fix& operator *= (int)
Fix& operator /= (const Fix&)
Fix operator + (const Fix&, const Fix&)
Fix operator - (const Fix&, const Fix&)
Fix operator * (const Fix&, const Fix&)
Fix operator * (const Fix&, int)
Fix operator * (int, const Fix&)
Fix operator / (const Fix&, const Fix&)
Fix operator - (const Fix&) // unary minus
int operator == (const Fix& a, const Fix& b)
int operator != (const Fix& a, const Fix& b)
int operator >= (const Fix& a, const Fix& b)
int operator <= (const Fix& a, const Fix& b)
int operator > (const Fix& a, const Fix& b)
int operator < (const Fix& a, const Fix& b)
```

Notes:

- These operators are designed so that overflow does not, as a rule, occur (the return value has a wider format than that of its arguments). The exception is when the result cannot be represented in a `Fix` with all 64 bits before the binary point.
- The output of any operation will have error codes that are the logical OR of those of the arguments to the operation, plus any additional errors that occurred during the operation (like divide by zero).
- The division operation is currently a cheat: it converts to double and computes the result, converting back to `Fix`.
- The relational operators `==`, `!=`, `>=`, `<=`, `>`, `<` are all written in terms of a function:

`int compare(const Fix& a, const Fix& b)`

Returns -1 if $a < b$, 0 if $a = b$, and 1 if $a > b$. The comparison is exact (every bit is checked) if the two values have the same precision format. If the precisions are different, the arguments are converted to doubles and compared. Since *double* values only have an accuracy of about 53 bits on most machines, this may cause false equality reports for *Fix* values with many bits.

Conversions

`operator int() const`

Returns the value of the *Fix* number as an integer, truncating towards zero.

`operator float() const`

`operator double() const`

Converts to a float or a double, creating an exact result when possible.

`void complement ()`

Replaces the current value by its complement.

Fix Overflow, Rounding, and Errors

The *Fix* class defines the following enumerated values for overflow handling:

`Fix::ovf_saturate`

`Fix::ovf_zero_saturate`

`Fix::ovf_wrapped`

`Fix::ovf_warning`

These can be used as arguments to the `set_overflow` method, as in the following example:

```
out.set_overflow(Fix::ovf_saturate)
```

The member function

`int overflow() const`

returns the overflow type. This returned result can be compared against the above enumerated values. Overflow types may also be specified as strings, using the following method:

```
void set_ovflow(const char* overflow_type);
the overflow_type argument may be one of "saturate", "zero_saturate",
"wrapped", or "warning".
```

The rounding behavior of a Fix value may be set by calling:

```
void set_rounding(int value);
```

If the argument is false, or has the value *Fix::mask_truncate*, truncation will occur. If the argument is nonzero (for example, if it has the value *Fix::mask_truncate_round*), rounding will occur. The older name *Set_MASK* is a synonym for *set_rounding*.

The following functions access the error bits of a Fix result:

```
int ovf_occurred() const
int invalid() const
int dbz() const
```

The first function returns TRUE if there have been any overflows in computing the value. The second returns TRUE if the value is invalid, because of invalid precision parameters or a divide by zero. The third returns TRUE only for divide by zero.

Defining New Data Types

The ADS Ptolemy heterogeneous message interface provides a mechanism for stars to transmit arbitrary objects to other stars. With this interface:

- You can safely modify large messages without excessive memory allocation and de-allocation.
- You may copy large messages by using a reference count mechanism, as in many C++ classes (for example, string classes).
- You may allocate existing stars to handle ANYTYPE message particles without change.
- You can define your own message types with relative ease; no change to the kernel is required to support new message types.

The Message type is understood by ADS Ptolemy to mean a particle containing a message. There are three classes that implement the support for message types:

Message

The *Message* class is the base class from which all other message data types are derived. A user wishing to define an application-specific message type derives a new class from *Message*.

Envelope

The *Envelope* class contains a pointer to a "derived from" *Message*. When an Envelope object is copied or duplicated, the new envelope simply sets its own pointer to the pointer contained in the original. Several envelopes can thus reference the same Message object. Each Message object contains a reference count, which tracks how many Envelope objects reference it; when the last reference is removed, the Message is deleted.

MessageParticle

The *MessageParticle* class is a type of *Particle* (like *IntParticle*, *FloatParticle*, etc.); it contains an *Envelope*. Ports of type *Message* transmit and receive objects of this type.

Class *Particle* contains five member functions for message support:

`void getMessage(const Envelope&)`

Receives a message.

`void accessMessage(const Envelope&)const`

Accesses the message, but does not remove it from the message particle.

`<< operator(const Envelope&)`

Loads an envelope's message into a particle.

`<< operator(Message&)`

Loads a message into a particle.

`int isMessage() const`

Returns TRUE if particle is a message.

The first four functions return errors in the base class; they are overridden in the *MessageParticle* class with functions that perform the expected operation.

Defining a New Message Class

Every user-defined message is derived from class *Message* and must be placed in the *AgilentPtolemy* namespace. Certain virtual functions defined in that class must be overridden; others may optionally be overridden. The following is an example of a user-defined message type (see `modelbuilder/examples/adsptolemy/message/Vector.h` under your ADS installation directory):

```
#ifndef VECTOR_H_INCLUDED
#define VECTOR_H_INCLUDED
// Not required, unless compiling under g++. These directives will
// not effect other compilers.
#ifdef __GNUG__
#pragma interface
#endif
#include "Message.h"
#include "agilent_vectorDll.h"
/* Data type of the new message type. This variable name must be in
   all capital letters and placed in the AgilentPtolemy namespace.
   The ptlang preprocessor will convert the port type field into all
   capital letters.*/
```


User-Defined Models

```

namespace AgilentPtolemy {
DllImport extern const AgilentPtolemy:: DataType VECTOR;
}
// A vector of doubles. Example of a user defined data type in ADS
// Ptolemy.
class Vector:public Message {
public:
    // Default Constructor
    Vector();
    // Copy Constructor
    Vector(const Vector&);
    // Destructor
    ~Vector();
    // Return the data type of the Message
    AgilentPtolemy:: DataType type() const {
        return AgilentPtolemy::VECTOR;
    }
    // Dynamically allocate a Vector identical to this one
    Message* clone() const {
        Vector* newMessage = new Vector(*this);
        return newMessage;
    }
    // Output the data structure as a string
    StringList print() const;
    /***** Optional Type Conversion to Scalar *****/
    // Return the Norm as float
    operator int() const { return (int)norm(); }
    // Return the Norm as float
    operator Fix() const { return norm(); }
    // Return the Norm as float
    operator float() const { return float(norm()); }
    // Return the Norm as double
    operator double() const { return norm(); }
    // Return the Norm as Complex
    operator Complex() const { return norm(); }
    /***** Optional support for initializable delays *****/
    // Parse the init delay string
    void operator << (const StringState&);
    // Pass through methods for the other operators, otherwise c++
    // will hide the following methods
    //
    void operator << (int i) { ((Message&) *this) << i; }
    //
    void operator << (double i) { ((Message&) *this) << i; }
    //
    void operator << (const Complex& i) { ((Message&) *this) << i; }
    //
    void operator << (const Fix& i) { ((Message&) *this) << i; }
    /***** Vector methods *****/
    // Return the Norm
    double norm() const;
    // Access a member of the vector
    inline double& operator[] (int i) {
        return vector[i];
    }
    // Access a member of the vector, const version
    inline const double& operator[] (int i) const {
        return vector[i];
    }
    // Resize the vector to a given length
    inline void resize(int i) {
        delete [] vector;
        if (i>0) {
            vector = new double[i];
            sz = i;
        }
        else {
            vector = NULL;
            sz = 0;
        }
    }
    // Return the size of this vector
    inline int size() const { return sz; }
private:

```

```
// Vector data members
// Array containing the data
double *vector;
// The size of the array
int sz;
};
#endif /*VECTOR_H_INCLUDED*/
```

Example: Using the Vector Message Class in a Custom Model

To build a new model using the *Vector* message type example:

1. Create new directory for model development as outlined in *Building Signal Processing Models* (modbuild).
2. Copy the files SDFConst_V.pl, SDFVectToFloat.pl, SDFVectToMx.pl, Vector.h, Vector.cc, and make-defs from the directory

\$HPEESOF_DIR/modelbuilder/examples/adsptolemy/message

into the src directory of your model development area.

3. Compile the source code following the directions in *Building Signal Processing Models* (modbuild).

This message object can contain a vector of doubles of arbitrary length. Some functions in the class are arbitrary and you may define them in whatever way is most convenient; however, there are some requirements:

- The class must redefine the *type* method from class *Message*. This function returns a string identifying the message type. This string should be identical to the name of the class.
- The class must define a copy constructor, unless the default copy constructor generated by the compiler (which does memberwise copying) will do the job.
- The class must redefine the *clone* method of class *Message*. Given that the copy constructor is defined, the form shown in the example, where a new object is created with the *new* operator and the copy constructor, will suffice.

In addition, you may optionally define type conversion, initializable delay parsing and printing functions if they make sense. If a star that produces messages is connected to a star that expects integers (or floating values, or complex values), the appropriate type conversion function is called. The base class, *Message*, defines the virtual conversion functions *int()*, *float()*, and *complex()* and the printing method *print()* -see the file Vector.cc in the modelbuilder/examples/adsptolemy/message directory of your ADS installation for their exact types. The base class conversion functions assert a run-time error, and the default print function returns a *StringList* reading:

<type>: no print method

where type is whatever is returned by *type()*.

By redefining these methods, you can make it legal to connect a star that generates messages to a star that expects integer, floating, or complex particles, or you can connect

to a *Printer* star.

Use of the Envelope Class

The Envelope class references objects of class Message or its derived classes. Once a Message object is placed into an Envelope object, the Envelope takes over responsibility for managing its memory, that is, maintaining reference counts and deleting the message when it is no longer needed.

The constructor (which takes as its argument a reference to a Message), copy constructor, assignment operator, and destructor of *Envelope* manipulate the reference counts of the reference's Message object. Assignment simply copies a pointer and increments the reference count. When an Envelope destructor is called, the reference count of the Message object is decremented; if it becomes zero, the Message object is deleted. Because of this deletion, a Message must never be put inside an Envelope unless it was created with the *new* operator. Once a Message object is put into an Envelope it must never be explicitly deleted; it will "live" as long as there is at least one Envelope that contains it.

It is possible for an Envelope to be *empty*. If it is, the *empty* method will return *TRUE*, and the data field will be *NULL*.

The *type* method of Envelope returns the datatype of the contained Message object. To access the data, the following two methods are provided:

- The *myData* function returns a pointer to the contained Message-derived object.

Note The data pointed to by this pointer must not be modified, since other Envelope objects in the program may also contain it. If you convert its type, always make sure that the converted type is a pointer to *const* :

```
Envelope pkt;
(input%0).getMessage(pkt);
const MyMessageType& myMsg = *(const MyMessageType *)pkt.myData();
```

This ensures that the compiler will complain if you do anything illegal.

- The *writableCopy* function also returns a pointer to the contained object, but with a difference. If the reference count is one, the envelope is emptied (set to the dummy message) and the contents are returned. If the reference count is greater than one, a clone of the contents is made (by calling its *clone()* function) and returned; again the envelope is zeroed (to prevent the making of additional clones later on).

In some cases, a star writer will need to keep a received Message object around between executions. The best way to do this is to have the star contain a member of type Envelope, and to use this member object to hold the message data between executions. Messages should always be kept in envelopes so that you do not have to worry about managing their memory.

Use of the MessageParticle Class

If a porthole is of type `Message`, then its particles are objects of the class `MessageParticle`. A `MessageParticle` is simply a particle whose data field is an `Envelope`, meaning that it can hold a `Message` in the same way that `Envelope` objects do.

The principal operations on `MessageParticle` objects are `<<` with an argument of either type *Envelope* or *Message* to load a message into the particle, and *getMessage(Envelope&)* to transfer message contents from the particle into a user-supplied message. The *getMessage* method removes the message contents from the particle.

This "aggressive reclamation" policy (both here and in other places) eliminates references to `Message` objects as soon as possible, thus minimizing the number of no-longer-needed messages in the system and preventing writable `Copy()` from generating unnecessary clones.

In cases where the destructive behavior of *getMessage* cannot be tolerated, an alternative interface, *accessMessage(Envelope&)*, is provided. The *accessMessage(Envelope&)* interface does not remove the message contents from the particle. Therefore, heavy use of *accessMessage* in systems where large-sized messages may be present can cause the amount of occupied virtual memory to grow (though all message will eventually be deleted).

The Matrix Data Types

The `PtMatrix` class is the primary support for matrix types in ADS Ptolemy. `PtMatrix` is derived from the `Message` class, and uses the various kernel support functions for working with the `Message` data type as described in the previous section, [Defining New Data Types](#).

This section describes the `PtMatrix` class and its use in writing stars and programs.

Design Philosophy

The `PtMatrix` class implements two dimensional arrays. Four key classes are derived from `PtMatrix`: *ComplexMatrix*, *FixMatrix*, *FloatMatrix*, and *IntMatrix*. (Note that *FloatMatrix* is a matrix of C++ *double* s.)

A survey of the matrix classes implemented by programmers revealed two primary styles of implementation: a vector of vectors and a simple array. Also highlighted were two entry storage formats: column-major ordering, where all the entries in the first column are stored before those of the second column, and row-major ordering, where the entries are stored row-by-row, starting with the first row. Column-major ordering is how Fortran stores arrays, whereas row-major ordering is how C stores arrays.

The ADS Ptolemy `PtMatrix` class stores data as a simple C array, and therefore uses row-major ordering. Row-major ordering also seems more sensible for operations such as

image and video processing, though it might make it more difficult to interface ADS Ptolemy's PtMatrix class with Fortran library calls. The limits of interfacing ADS Ptolemy's PtMatrix class with other software is discussed in the section [Public Functions and Operators for the PtMatrix Class](#).

The decision to store data entries in a C array rather than as an array of vector objects resulted in a greater effect on performance than that of using row-major versus column-major ordering. Implementing a matrix class as an array of vector class objects has a couple of advantages: referencing an entry may be faster, and it is easier to do operations on a whole row or column of the matrix, depending on whether the format is an array of column vectors or an array of row vectors.

An entry lookup in an array of row vectors requires two index lookups: one to find the desired row vector in the array, and one to find the desired entry in that row. A linear array, by contrast, requires a multiplication to find the location of the first element of the desired row, and then an index lookup to find the column in that row. For example, $A[row][col]$ is equivalent to looking up $\&data + (row * numRows + col)$ if the entries are stored in a C array $data[]$, whereas it is $\&rowArray + row + col$ if looking up the entry in an array-of-vectors format. Although the array of vectors format has faster lookups, it is also more expensive to create and delete the matrix. Each vector of the array must be created in the matrix constructor, and then each vector must be deleted by the matrix destructor. The array of vectors format also requires more memory to store the data and the extra array of vectors.

Given the advantages and disadvantages of the two systems, the PtMatrix class was designed to store data in a standard C array. ADS Ptolemy's environment is such that matrices are constantly created and deleted as needed by stars; this negates much of the speed gained from faster lookups.

The PtMatrix Class

The PtMatrix base class is derived from the Message class so that you can use ADS Ptolemy's Envelope class and message-handling system.

As explained previously, there are currently four data-specific matrix classes: ComplexMatrix, FixMatrix, FloatMatrix, and IntMatrix. Each of these classes stores its entries in a standard C array named *data*, which is an array of data objects corresponding to the PtMatrix type: *Complex*, *Fix*, *double*, or *int*. These four matrix classes implement a common set of operators and functions; in addition, the ComplexMatrix class has a few special methods such as *conjugate()* and *hermitian()*, and the FixMatrix class has a number of special constructors that allow you to specify the precision of the entries in the matrix. Generally, all entries of a FixMatrix will have the same precision.

The matrix classes were designed to take full advantage of operator overloading in C++ so that operations on matrix objects can be written much like operations on scalar objects. For example, the two-operand multiply *operator* $*$ has been defined so that if *A* and *B* are matrices, $A * B$ will return a third matrix that is the matrix product of *A* and *B*.

Public Functions and Operators for the PtMatrix Class

The functions and operators listed below are implemented by all matrix classes (ComplexMatrix, FixMatrix, FloatMatrix, and IntMatrix) unless otherwise noted. The symbols used are:

XXX which refers to one of *Complex*, *Fix*, *Float*, or *Int*

xxx which refers to one of *Complex*, *Fix*, *double*, or *int*

Functions and Operators to Access Entries of the Matrix

xxx& entry(int i)

Example: A.entry(i).

Return the *ith* entry of the matrix when its data storage is a linear array. This facilitates quick operations on every entry of the matrix without regard for the specific (row, column) position of that entry. The total number of entries in the matrix is defined to be numRows() * numCols(), with indices ranging from 0 to numRows() * numCols() - 1. This function returns a reference to the actual entry in the matrix so that assignments can be made to that entry. In general, functions intended to linearly reference each entry of a matrix A should use this instead of the expression A.data[i] because classes derived from PtMatrix can then overload the entry() method and reuse the same functions.

xxx* operator [] (int row)

Example: A[row][column].

Return a pointer to the start of the row in the matrix's data storage. (This operation is different from that of matrix classes defined as arrays of vectors, in which the [] operator returns the vector representing the desired row.) This operator is generally not used alone, rather it is used with the [] operator defined on C arrays so that A[i][j] will give you the entry of the matrix in the *ith* row and *jth* column of the data storage. The range of rows is from 0 to numRows()-1 and the range of columns is from 0 to numCols()-1.

Constructors

XXXMatrix()

Example: IntMatrix A.

Create an uninitialized matrix. Row and column numbers are set to zero and no memory is allocated for data storage.

XXXMatrix(int numRows, int numCol)

Example: FloatMatrix A(3,2).

Create a matrix with dimensions numRow by numCol. Memory is allocated for data storage but the entries are uninitialized.

`XXXMatrix(int numRow, int numCol, PortHole& p)`

Example: `ComplexMatrix(3,3,myPortHole)`.

Create a matrix of the given dimensions and initialize the entries by assigning to them values taken from the porthole myPortHole. The entries are assigned in a rasterized sequence so that the value of the first particle removed from the porthole is assigned to entry (0,0), the second particle's value to entry (0,1), etc. It is assumed that the porthole has enough particles in its buffer to fill all the entries of the new matrix.

`XXXMatrix(int numRow, int numCol, XXXArrayState& dataArray)`

Example: `IntMatrix A(2,2,myIntArrayState)`.

Create a matrix with the given dimensions and initialize the entries to the values in the given ArrayState. The values of the ArrayState fill the matrix in rasterized sequence so that entry (0,0) of the matrix is the first entry of the ArrayState, entry (0,1) of the matrix is the second, etc. An error is generated if the ArrayState does not have enough values to initialize the whole matrix.

`XXXMatrix(const XXXMatrix& src)`

Example: `FixMatrix A(B)`.

This is the copy constructor. A new matrix is formed with the same dimensions as the source matrix and the data values are copied from the source.

`XXXMatrix(const XXXMatrix& src, int startRow, int startCol, int numRow, int numCol)`

Example: `IntMatrix A(B,2,2,3,3)`.

This special "submatrix" constructor creates a new matrix whose values come from a submatrix of the source. The arguments startRow and startCols specify the starting row and column of the source matrix. The values numRow and numCol specify the dimensions of the new matrix. The sum startRow + numRow must not be greater than the maximum number of rows in the source matrix; similarly, startCol + numCol must not be greater than the maximum number of columns in the source. For example, if B is a matrix with dimension (4,4), then `A(B,1,1,2,2)` would create a new matrix A that is a (2,2) matrix with data values from the center quadrant of matrix B, so that `A[0][0] == B[1][1]`, `A[0][1] == B[1][2]`, `A[1][0] == B[2][1]`, and `A[1][1] == B[2][2]`.

The following are special constructors for the *FixMatrix* class that enable the programmer to specify the precision of the *FixMatrix* entries.

`FixMatrix(int numRow, int numCol, int length, int intBits)`

Example: `FixMatrix A(2,2,14,4)`.

Create a FixMatrix with the given dimensions such that each entry is a fixed-point number with precision as given by the length and intBits inputs.

`FixMatrix(int numRows, int numCol, int length, int intBits, PortHole& myPortHole)`

Example: `FixMatrix A(2,2,14,4).`

Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` inputs and initialized with the values that are read from the particles contained in the porthole `myPortHole`.

`FixMatrix(int numRows, int numCol, int length, int intBits, FixArrayState& dataArray)`

Example: `FixMatrix A(2,2,14,4).`

Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` inputs and initialized with the values in the given `FixArrayState`.

There are also special copy constructors for the `FixMatrix` class that enable the programmer to specify the precision of the entries of the `FixMatrix` as they are copied from the sources. These copy constructors are usually used for easy conversion between the other matrix types. The last argument specifies the type of masking function (truncate, rounding, etc.) to be used when doing the conversion.

`FixMatrix(const XXXMatrix& src, int length, int intBits, int round)`

Example: `FixMatrix A(CxMatrix,4,14,TRUE).`

Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the `length` and `intBits` arguments. Each entry of the new matrix is copied from the corresponding entry of the `src` matrix and converted as specified by the `round` argument.

Comparison Operators

`int operator == (const XXXMatrix& src)`

Example: `if(A == B) then ...`

Return `TRUE` if the two matrices have the same dimensions and every entry in `A` is equal to the corresponding entry in `B`. Return `FALSE` otherwise.

`int operator != (const XXXMatrix& src)`

Example: `if(A != B) then ...`

Return `TRUE` if the two matrices have different dimensions or if any entry of `A` differs from the corresponding entry in `B`. Return `FALSE` otherwise.

Conversion Operators

Each matrix class has a conversion operator so that the programmer can explicitly cast

one type of matrix to another (this casting is done by copying). It would have been possible to make conversions occur automatically when needed, but because these conversions can be quite expensive for large matrices, and because unexpected results might occur if the user did not intend for a conversion to occur, we chose to require that these conversions be used explicitly.

operator XXXMatrix () const

Example: FloatMatrix =A * (FloatMatrix)B.

Convert a matrix of one type into another. These conversions allow the various arithmetic operators, such as * and +, to be used on matrices of different types. For example, if A in the example above is a (3,3) FloatMatrix and B is a (3,2) IntMatrix, then C is a FloatMatrix with dimensions (3,2).

Destructive Replacement Operators

These operators are member functions that modify the current value of the object. In the following examples, A is usually the lvalue (**this*). All operators return **this*:

XXXMatrix& operator = (const XXXMatrix& src)

Example: A = B.

This is the assignment operator: make A into a matrix that is a copy of B. If A already has allocated data storage, then the size of this data storage is compared to the size of B. If they are equal, then the dimensions of A are simply set to those of B and the entries copied. If they are not equal, the data storage is freed and reallocated before copying.

XXXMatrix& operator = (xxx value)

Example: A = value.

Assign each entry of A to have the given value. Memory management is handled as in the previous operator. Warning: this operator is targeted for deletion. Do not use it.

XXXMatrix& operator += (const XXXMatrix& src)

Example: A += B.

Perform the operation $A.entry(i) += B.entry(i)$ for each entry in A. A and B must have the same dimensions.

XXXMatrix& operator += (xxx value)

Example: A += value.

Add the scalar value to each entry in the matrix.

XXXMatrix& operator -= (const XXXMatrix& src)

Example: A -= B.

Perform the operation $A.\text{entry}(i) -= B.\text{entry}(i)$ for each entry in A. A and B must have the same dimensions.

`XXXMatrix& operator -= (xxx value)`

Example: $A -= \text{value}$.

Subtract the scalar value from each entry in the matrix.

`XXXMatrix& operator *= (const XXXMatrix& src)`

Example: $A *= B$.

Perform the operation $A.\text{entry}(i) *= B.\text{entry}(i)$ for each entry in A. A and B must have the same dimensions. Note: this is an elementwise operation and is not equivalent to $A = A * B$.

`XXXMatrix& operator *= (xxx value)`

Example: $A *= \text{value}$.

Multiply each matrix entry by the scalar value.

`XXXMatrix& operator /= (const XXXMatrix& src)`

Example: $A /= B$.

Perform the operation $A.\text{entry}(i) /= B.\text{entry}(i)$ for each entry in A. A and B must have the same dimensions.

`XXXMatrix& operator /= (xxx value)`

Example: $A /= \text{value}$.

Divide each matrix entry by the scalar value. This value must be non-zero.

`XXXMatrix& operator identity()`

Example: $A.\text{identity}()$.

Change A to be an identity matrix so that each entry on the diagonal is 1 and all off-diagonal entries are 0.

Non-Destructive Operators (These Return a New Matrix)

`XXXMatrix operator - ()`

Example: $B = -A$.

Return a new matrix such that each element is the negative of the source element.

`XXXMatrix operator ~ ()`

Example: $B = \sim A$.

Return a new matrix that is the transpose of the source.

XXXMatrix operator ! ()

Example: $B = !A$.

Return a new matrix that is the inverse of the source.

XXXMatrix operator ^ (int exponent)

Example: $B = A^2$.

Return a new matrix that is the source matrix to the given exponent power. The exponent can be negative, in which case it is first treated as a positive number and the final result is then inverted. So $A^2 == A * A$ and $A^{-3} == !(A * A * A)$.

XXXMatrix transpose()

Example: $B = A.\text{transpose}()$.

This is the same as the \sim operator but called by a function name instead of as an operator.

XXXMatrix inverse()

Example: $B = A.\text{inverse}()$.

This is the same as the $!$ operator but called by a function name instead of as an operator.

ComplexMatrix conjugate()

Example: $\text{ComplexMatrix } B = A.\text{conjugate}()$.

Return a new matrix such that each element is the complex conjugate of the source. This function is defined for the ComplexMatrix class only.

ComplexMatrix hermitian()

Example: $\text{ComplexMatrix } B = A.\text{hermitian}()$.

Return a new matrix that is the Hermitian Transpose (conjugate transpose) of the source. This function is defined for the ComplexMatrix class only.

Non-Member Binary Operators

XXXMatrix operator + (const XXXMatrix& left, const XXXMatrix& right)

Example: $A = B + C$.

Return a new matrix that is the sum of the first two. The left and right source matrices must have the same dimensions.

XXXMatrix operator + (const xxx& scalar, const XXXMatrix& matrix)

Example: $A = 5 + B$.

Return a new matrix that has entries of the source matrix added to a scalar

value.

`XXXMatrix operator + (const XXXMatrix& matrix, const xxx& scalar)`

Example: $A = B + 5$.

Return a new matrix that has entries of the source matrix added to a scalar value. (This is the same as the previous operator but with the scalar on the right.)

`XXXMatrix operator - (const XXXMatrix& left, const XXXMatrix& right)`

Example: $A = B - C$.

Return a new matrix that is the difference of the first two. The left and right source matrices must have the same dimensions.

`XXXMatrix operator - (const xxx& scalar, const XXXMatrix& matrix)`

Example: $A = 5 - B$.

Return a new matrix that has the negative of the entries of the source matrix added to a scalar value.

`XXXMatrix operator - (const XXXMatrix& matrix, const xxx& scalar)`

Example: $A = B - 5$.

Return a new matrix such that each entry is the corresponding entry of the source matrix minus the scalar value.

`XXXMatrix operator * (const XXXMatrix& left, const XXXMatrix& right)`

Example: $A = B * C$.

Return a new matrix that is the matrix product of the first two. The left and right source matrices must have compatible dimensions; for example, `A.numCols() == B.numRows()`.

`XXXMatrix operator * (const xxx& scalar, const XXXMatrix& matrix)`

Example: $A = 5 * B$.

Return a new matrix that has entries of the source matrix multiplied by a scalar value.

`XXXMatrix operator * (const XXXMatrix& matrix, const xxx& scalar)`

Example: $A = B * 5$.

Return a new matrix that has entries of the source matrix multiplied by a scalar value. (This is the same as the previous operator, but with the scalar value on the right.)

Miscellaneous Functions

```
int numRows()
```

Return the number of rows in the matrix.

```
int numCols()
```

Return the number of columns in the matrix.

```
Message* clone()
```

Example: `IntMatrix *B = A.clone()`.

Return a copy of `*this`.

```
StringList print()
```

Example: `A.print()`.

Return a formatted `StringList` that can be printed to display the contents of the matrix in a reasonable format.

```
XXXMatrix& multiply (const XXXMatrix& left, const XXXMatrix& right, XXXMatrix& result)
```

Example: `multiply(A,B,C)`.

This is a faster 3-operand form of matrix multiply such that the result matrix is passed as an argument in order to avoid the extra copy step involved when writing `C = A * B`.

```
const char* dataType()
```

Example: `A.dataType()`.

Return a string that specifies the name of the matrix type. Available strings are `ComplexMatrix`, `FixMatrix`, `FloatMatrix`, and `IntMatrix`.

```
int isA(const char* type)
```

Example: `if(A.isA("FixMatrix")) then ...`

Return `TRUE` if the argument string matches the type string of the matrix.

Writing Stars and Programs Using the PtMatrix Class

This section describes how to use the matrix data classes when writing stars. Some examples are given here. For more examples, refer to the stars in `$PTOLEMY/src/domains/sdf/matrix/stars/*.pl` and `$PTOLEMY/src/domains/sdf/image/stars/*.pl`.

Memory Management

The most important thing to understand about the use of matrix data classes in the ADS

Ptolemy environment is that stars designated to output the matrix in a particle should allocate memory for the matrix, *but never delete that matrix*. Strange errors occur if the star deletes the matrix before it is used by another star later in the execution sequence. Memory reclamation is automatically performed by the reference-counting mechanism of the Message class.

Naming Conventions

Stars that implement general-purpose matrix operations usually have names with the *_M* suffix to distinguish them from stars that operate on scalar particles. For example, the *SDFGain_M* star multiplies an input matrix by a scalar value and outputs the resulting matrix. This is in contrast to *SDFGain*, which multiplies an input value held in a FloatParticle by a double and puts that result in an output FloatParticle.

Include Files

For a star to use the PtMatrix classes, it must include the file *Matrix.h* in either its *.h* or *.cc* file. If the star has a matrix data member, then the declaration

```
hinclude {"ADSPtolemy Matrix.h"}
```

needs to be in the Star definition. Otherwise, the declaration

```
ccinclude {"ADSPtolemyMatrix.h"}
```

is sufficient.

To declare an input porthole that accepts matrices, the following syntax is used:

```
input {
  name {inputPortHole}
  type {FLOAT_MATRIX}
}
```

The syntax is the same for output portholes. The type field can be `COMPLEX_MATRIX`, `FLOAT_MATRIX`, `FIX_MATRIX`, or `INT_MATRIX`.

The icons created by ADS Ptolemy will have terminals that are thicker and have larger arrow points than the terminals for scalar particle types. The terminal colors follow the pattern of colors for scalar data types (for example, blue represents Float and FloatMatrix).

Input Portholes

The syntax for extracting a matrix from the input porthole is:

```
Envelope inPkt;
(inputPortHole%0).getMessage(inPkt);
const FloatMatrix& inputMatrix =
    *(const FloatMatrix *)inPkt.myData();
```

The first line declares an `Envelope`, used to access the matrix. (For more details on the `Envelope` class, see [Use of the Envelope Class](#).) The second line fills the envelope with the input matrix. Note that, because of the reference-counting mechanism, this line does not make a copy of the matrix. The last two lines extract a reference to the matrix from the envelope. It is up to the programmer to ensure that the cast agrees with the input port definition.

Because multiple envelopes might reference the same matrix, a star is generally not permitted to modify the matrix held by the `Envelope`. Thus, the function `myData()` returns a *const Message **. This is cast to be a *const FloatMatrix ** and then de-referenced and the value is assigned to `inputMatrix`. It is generally better to handle matrices by reference rather than pointer because it is clearer to write $A + B$ rather than $*A + *B$ when working with matrix operations. Stars that modify an input matrix should access it using the `writableCopy` method, as explained in [Use of the Envelope Class](#).

Allowing Delays on Inputs

The cast to *(const FloatMatrix *)* above is not always safe. Even if the source star is known to provide matrices of the appropriate type, a delay on the arc connecting the two stars can cause problems. In particular, delays in dataflow domains are implemented as initial particles on the arcs. These initial particles receive the value zero as defined by the type of particle. For `Message` particles, a zero is a `Message` particle containing a copy of the prototype `Message` registered with `RegisterMessage` (see the example `Vector.cc` file in [Defining a New Message Class](#)).

Matrix Outputs

To put a matrix into an output porthole , the syntax is:

```
FloatMatrix& outMatrix =*(new FloatMatrix(someRow,someCol));
// ... do some operations on the outMatrix
outputPortHole%0 << outMatrix;
```

The last line is similar to that for outputting a scalar value. This is because we have overloaded the `<< operator` for `MatrixEnvParticles` to support `PtMatrix` class inputs. The standard use of the `MessageParticle` class requires you to put your message into an envelope first and then use `<<` on the envelope (see [Use of the Envelope Class](#)), but we have specialized this so that the extra operation of creating an envelope first is not explicit.

The following is an example of a complete star definition that inputs and outputs `{{}}`

matrices:

```
defstar {
  name {Add_M}
  domain {SDF}
  desc {Output is the sum of all the floating-point input matrices.}
  location {Numeric Matrix}
  inmulti {
    name {input}
    type {FLOAT_MATRIX}
  }
  output {
    name {output}
    type {FLOAT_MATRIX}
  }
  ccinclude {"ADSPtolemyMatrix.h"}
  go {
    // set up the multi-port
    MPHIter nexti(input);
    // Get the first input matrix
    PortHole *p = nexti++;
    Envelope firstPkt;
    ((*p)%0).getMessage(firstPkt);
    const FloatMatrix& firstMatrix =
    *(const FloatMatrix*)firstPkt.myData();
    FloatMatrix& result = *(new
    FloatMatrix(firstMatrix.numRows(),firstMatrix.numCols()));
    result = firstMatrix;
    // Add in the remaining inputs
    while ((p = nexti++) != 0)
    {
      Envelope nextPkt;
      ((*p)%0).getMessage(nextPkt);
      const FloatMatrix& nextMatrix = *(const
      FloatMatrix*)nextPkt.myData();
      result += nextMatrix;
    }
    // Send out finished result
    output%0 << result;
  }
}
```

Writing Stars That Manipulate Any Particle Type

In ADS Ptolemy, stars can declare inputs and outputs of type *ANYTYPE*. A star may need to do this, for example, if it simply copies its inputs without regard to type, as in the case of a Fork star, or if it calls a generic function that is overloaded by every data type, such as sink stars which call the print method of the type.

The following example illustrates a star that operates on ANYTYPE particles:

```
defstar {
  name {Fork}
  domain {SDF}
  desc {Copy input particles to each output.}
  input {
    name{input}
    type{ANYTYPE}
  }
  outmulti {
```



```

    name{output}
    type{= input}
}
go {
    MPHIter nextp(output);
    PortHole* p;
    while ((p = nextp++) != 0)
        (*p)%0 = input%0;
}
}

```

Notice how, in the definition of the output type, the star simply says that its output type will be the same as the input type.

Timed Particle Signal Type

ADS Ptolemy supports timed data. This signal is derived from complex data and includes additional attributes. The timed signal packet includes five members:

$\{i(t), q(t), \text{flavor}, F_c, \text{and } t\}$

where $i(t)$ and $q(t)$ are the timed signal in phase and quadrature components, *flavor* represents a modulated signal, F_c is the carrier (or characterization) frequency, and t is the time.

There are two equivalent representations (*flavors*) of a timed signal:

complex envelope (ComplexEnv) $v(t)$
 real baseband (BaseBand) $V(t)$

RF signals that are represented in the ComplexEnv flavor $v(t)$ together with F_c can be converted to the real BaseBand flavor $V(t)$ as:

$$V(t) = \text{Re} \left\{ v(t) e^{j2\pi F_c t} \right\}$$

Constructors

TimedParticle(const Complex& cx, const double& t, const double& fc);

Creates a Timed particle with a cx data, at time t, centered around F_c . Flavor is set to Complex Envelope.

TimedParticle(const double& x, const double& t);

Creates a Timed particle with data x, at time t. The F_c is set to zero and the flavor to Baseband.

TimedParticle(const int& x, const double& t);

Creates a Timed particle with data x , at time t . The Fc is set to zero and the flavor to Baseband.

`TimedParticle(const Fix& x, const double& t);`

Creates a Timed particle with data x , at time t . The Fc is set to zero and the flavor to Baseband.

`TimedParticle();`

Creates a Timed particle with data (0.0, 0.0), at time $t=0$. The Fc is set to zero and the flavor to Baseband.

Conversion Operators

`TimedParticle::operator int () const`

Returns the baseband representation of the timed data when the flavor is Complex Envelope, or just the data if the flavor is Baseband. The result is then converted to int.

`TimedParticle::operator Fix () const`

Returns the baseband representation of the timed data when the flavor is Complex Envelope, or just the data if the flavor is Baseband. The result is then converted to Fix.

`TimedParticle::operator float () const`

Returns the baseband representation of the timed data when the flavor is Complex Envelope, or just the data if the flavor is Baseband. The result is then converted to float.

`TimedParticle::operator double () const`

Returns the baseband representation of the timed data when the flavor is Complex Envelope, or just the data when the flavor is Baseband. The result is then converted to double.

`TimedParticle::operator complex () const`

Returns the (complex) data if the flavor is Complex Envelope, or just the real part of data when the flavor is Baseband. The result is then converted to complex.

`StringList TimedParticle::print () const`

Returns the (real and imaginary part of) data, flavor, time and carrier frequency associated with the timed particle.

Particle& TimedParticle::initialize()

Initializes the data, time and carrier frequency to zero and sets the flavor to Baseband.

Loading Timed Particle with Data

```
void operator << (int arg);
```

Loads the arg in the timed port by setting data = arg, flavor = Baseband. The time and Fc members are set by the kernel.

```
void operator << (double f);
```

Loads the arg in the timed port by setting data = arg, flavor = Baseband. The time and Fc members are set by the kernel.

```
void operator << (const Complex& c);
```

Loads the arg in the timed port by setting data = arg, flavor = ComplexEnv. The time and Fc members are set by the kernel.

```
void operator << (const Fix& x);
```

Loads the arg in the timed port by setting data = arg, flavor = Baseband. The time and Fc members are set by the kernel.

```
void operator << (const Envelope&);
```

This is not allowed. An error message is issued.

```
Particle& operator = (const Particle&);
```

Copies a timed particle into another one.

```
int operator == (const Particle&);
```

Compares all the members of the two timed particles delineated. Returns TRUE or FALSE.

Time Step Resolution

In a user-compiled model, to explicitly set a time step for any given port, you must insert code into the component's setup method to assign the value of the time step to the port object. The port object must be a single port object (not a multi-port object). Reference the following code fragment for the port object (called output) and the begin method as

you would code it in the component's ptlang file. This example also demonstrates how to use the setup method to set the output carrier frequency and how to use begin method to set the default time step available at the output port.

```

output {
  Name { out1 }
  Type { Timed }
  Desc { Timed source output signal }
}
defstate {
  name { TStep }
  type { float }
  default { 0.001 }
  desc { Simulation time step; use a value of 0 for time step synchronization with
  other network timed signals }
  units { TIME_UNIT }
  attributes { A_SETTABLE | A_NONCONSTANT }
}

defstate {
  name { FCarrier }
  type { float }
  default { 1000000 }
  desc { Output signal carrier frequency } units { FREQUENCY_UNIT }
  attributes { A_SETTABLE | A_NONCONSTANT }
}

setup {
  out1.setTimeStep( double(TStep));
  out1.setCarrierFrequency( double(FCarrier));
}

begin {
  if ( double(TStep) == 0.) {
    double timestep;
    timestep = out1.getTimeStep();
    TStep = timestep;
  }
  out1.setTimeStep( double(TStep));
}

```

Carrier Frequency Resolution

For Timed data types, many times the carrier frequency must be propagated from the inputs to the outputs.

Each Timed user-compiled component can define a custom way of propagating the carrier frequency from the inputs to the outputs. By default, each output is marked with the maximum F_c available at all of the inputs. To override this method, use the following ptlang template. The following template can be modified to meet the specific component

requirement for output carrier frequency:

```
method {
  name {propagateFc }
  access { protected }
  arglist { "(double*)" }
  type { void }
  code {
    // Create an iterator for the ports of this star
    BlockPortIter nextPort(*this);
    TSDFPortHole* port;
    // Find the maximum fc over all of the inputs
    double maxFc = 0;
    while ((port = (TSDFPortHole*)nextPort++) != NULL) {
      // Ignore unconnected ports and output ports
      if (!(port->far() || port->isItOutput())) continue;
      double fc = port->getCarrierFrequency();
      if (fc > maxFc) maxFc = fc;
    }
    // Reset the iterator
    nextPort.reset();
    // Now set all Timed output ports to the maximum Fc
    // found over all of the inputs
    while ((port = (TSDFPortHole*)nextPort++) != NULL) {
      // Ignore unconnected ports and input ports
      if (!(port->far() ||
              port->isItInput() ||
              (port->resolvedType() != TIMED)))
          continue;
      port->setCarrierFrequency(maxFc);
    }
  }
}
```

Porting UC Berkeley Ptolemy Models

A major design goal of ADS Ptolemy was to make it as backward compatible as possible with University of California, Berkeley (UCB) Ptolemy. We have only changed interfaces that were found to be not sufficiently robust or poorly implemented.

ADS Ptolemy currently supports most SDF UCB Ptolemy stars (component models). The porting process for one of these stars is relatively straight forward. In this section we will review the incompatibilities that require code modifications in the porting process of your star.

To begin your port, follow the directions to create a new model builder workspace outlined in the beginning of *Building Signal Processing Models* (modbuild). Edit the networks/user.mak and list the stars to be compiled in the USER_STARS variable field.

If you are porting any of the following three types of stars, follow the corresponding porting directions outlined in the sections below:

- Stars that use Tcl/Tk
- Stars making use of the Message Class
- Matrix stars

Once you are ready, compile and link your model as outlined in *Building Signal Processing Models* (modbuild).

Tcl/Tk Porting Issues

There are two differences in Tcl/Tk use in ADS Ptolemy vs. UCB Ptolemy. The first is that ADS Ptolemy uses Tcl/Tk 8.4 whereas UCB Ptolemy uses Tcl 7.6 / Tk 7.4 with iTcl extensions. There are many advantages in using Tcl/Tk 8.4 including native look and feel, speed improvements and improved widgets. Unfortunately, iTcl has not been ported to Tcl/Tk 8.x at this time. No ported UCB Ptolemy Tcl/Tk script can use iTcl extensions.

The second difference in the Tcl/Tk implementation of ADS Ptolemy is that pTcl is not included. pTcl allows a user to program a scripted run in UCB Ptolemy. The primary use for this function is to sweep simulation parameters or to perform a very simple optimization. ADS Ptolemy has its own methods for parameter sweeping and optimization, as described in earlier sections.

Porting Procedures

- Resolve all Tcl/Tk 8.4 vs. Tcl 7.6/Tk 7.4 issues. These should be minor, and apparent when you first run the Tcl/Tk script.
- Remove dependencies on iTcl. Most Tcl/Tk stars written for UCB Ptolemy do not make use of any iTcl, including all SDF Tcl/Tk stars shipped in the 0.7 release.
- Remove any calls to pTcl functions.

Message Class

This section assumes that you are both familiar with the Message class implementation in UCB Ptolemy and have read the ADS Ptolemy Message documentation in *Defining New Data Types* (modbuild) in *Data Types for Model Builders* (modbuild). The Message class infrastructure has been greatly simplified and enhanced in ADS Ptolemy. The improvements include:

- The new Message class can have types that are resolved by the type resolution algorithms. In UCB Ptolemy, all arcs that carried messages were resolved to the MESSAGE data type.
- Automatic type conversion between data types built into ADS Ptolemy and new message classes can be defined.
- A default *delay* message can be defined. In UCB Ptolemy, a delay message would have a DUMMY message. For more information, refer to *Defining New Data Types* (modbuild) in *Data Types for Model Builders* (modbuild).

An example vector Message class implementation complete with type conversion stars is provided in ADS Ptolemy. See the example in the section *Defining New Data Types* (modbuild) in *Data Types for Model Builders* (modbuild).

Porting Procedures

- Using the Vector message class example, modify your message class to conform to the new ADS Ptolemy Message class. Refer to *Defining a New Message Class* (modbuild) in *Data Types for Model Builders* (modbuild), for more information. This should be a relatively straight forward procedure.
- Optional: Search and replace the MESSAGE port data type in your star library with the data type appropriate for your message.
- Optional: Create type conversion stars, using the type conversion ptlang keyword in place of the defstar directive. This will enable ADS Ptolemy to do automatic type conversions.

Matrix Stars

The data type `_ENV` suffix has been dropped. Therefore, you must:

- Search and replace `FLOAT_MATRIX_ENV` with `FLOAT_MATRIX`.
- Search and replace `INT_MATRIX_ENV` with `INT_MATRIX`.
- Search and replace `FIX_MATRIX_ENV` with `FIX_MATRIX`.
- Search and replace `COMPLEX_MATRIX_ENV` with `COMPLEX_MATRIX`.

User-Defined Models API Reference

This section is provided as a reference and includes class descriptions, data structure diagrams, and application procedural interface (API) functions.

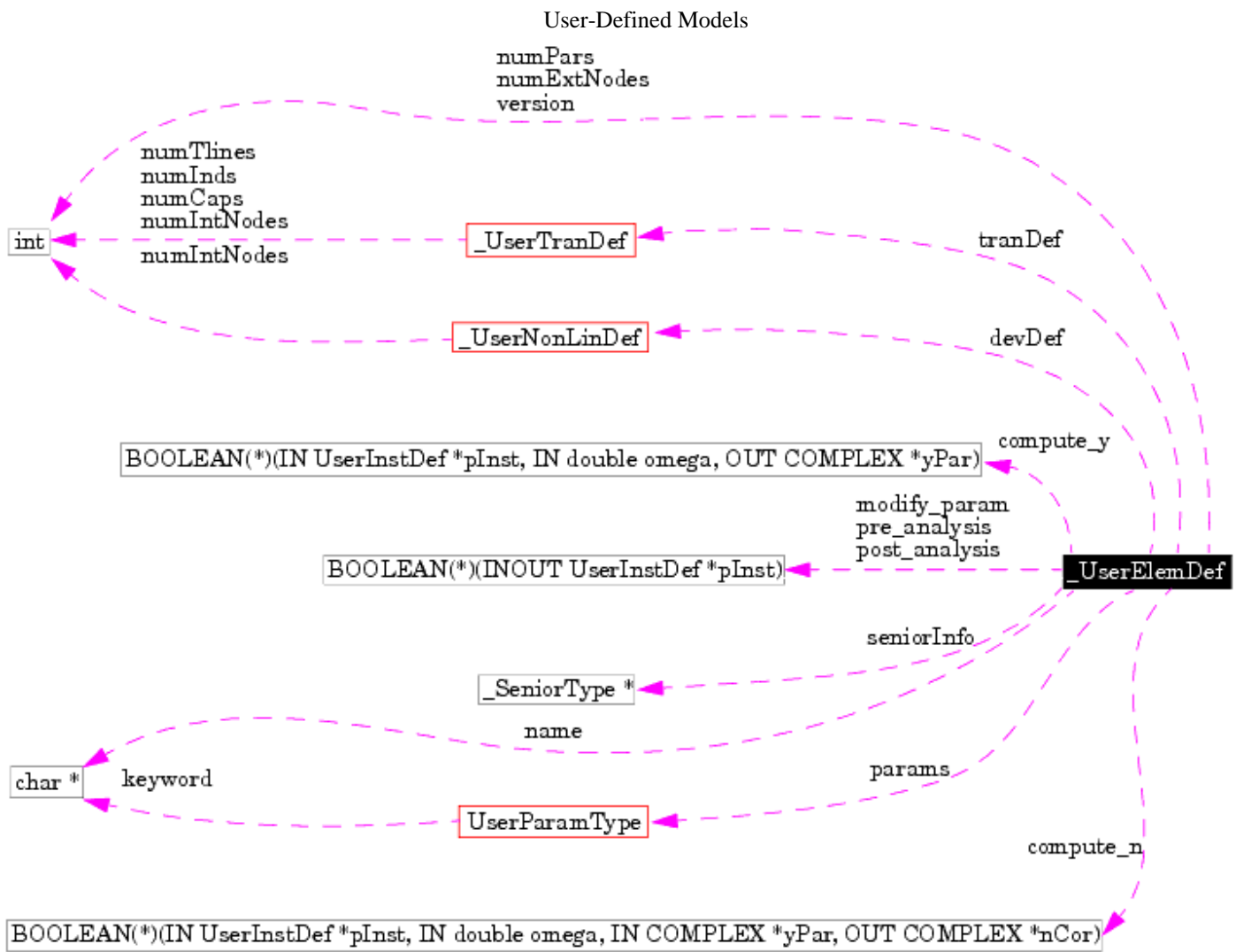
Class List

The table below shows a list of the structs with a brief description of each. For a detailed description of struct data members, refer to *Characteristics of User-Compiled Elements* (modbuild).

Class List

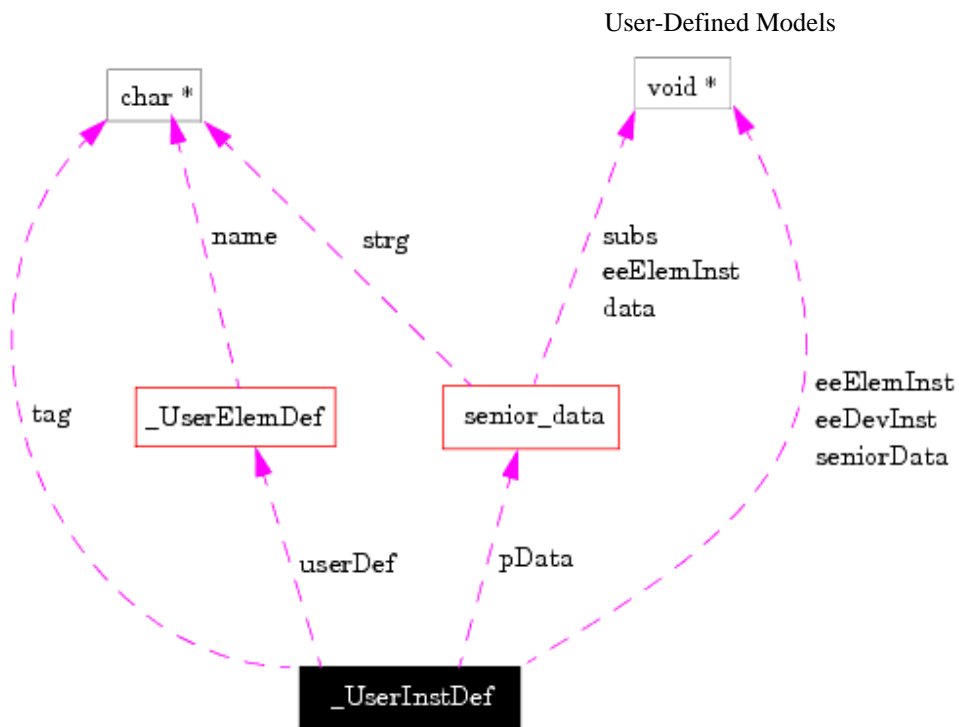
Struct	Description	Collaboration Diagrams
UserElemDef	Senior user element definition	UserElemDef Struct Reference
UserInstDef	The struct UserInstDef defines a senior user element instance	UserInstDef Struct Reference
UserNonLinDef	Senior nonlinear device definition A nonlinear element response is characterized by the functions in the UserNonLinDef type	UserNonLinDef Struct Reference
UserTranDef	Senior transient device definition	UserTranDef Struct Reference
COMPLEX	Linear response modeled in the frequency domain is complex, so the COMPLEX type is used for admittance (Y), scattering (S), and noise current-correlation parameters	COMPLEX Struct Reference
NParType	Conventional 2-port Noise parameters	NParType Struct Reference
senior_data	Parameter value of an instance. The parameter values of an item are obtained in an array of this data type.	senior_data Struct Reference
SENIOR_USER_DATA	Parameter value of an instance. SENIOR_USER_DATA is a typedef of struct senior_data	senior_data Struct Reference
UserParamType	Element parameter definition	UserParamType Struct Reference

UserElemDef Struct Reference



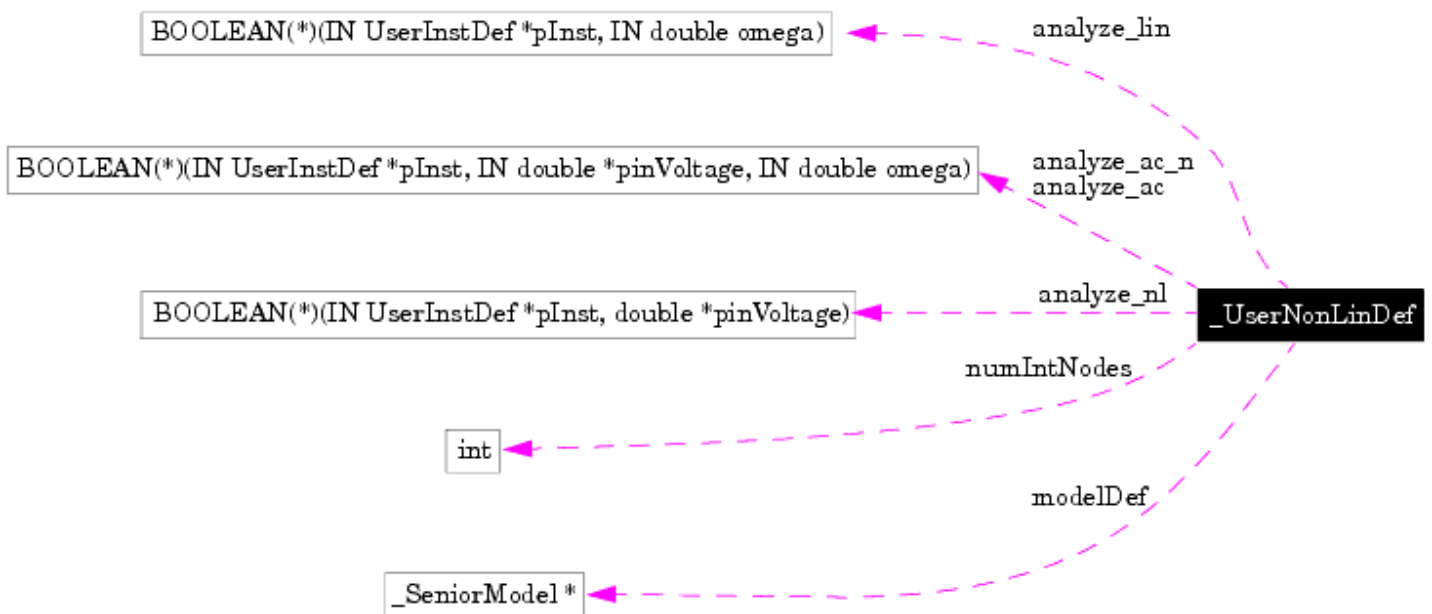
Collaboration Diagram for UserElemDef

UserInstDef Struct Reference



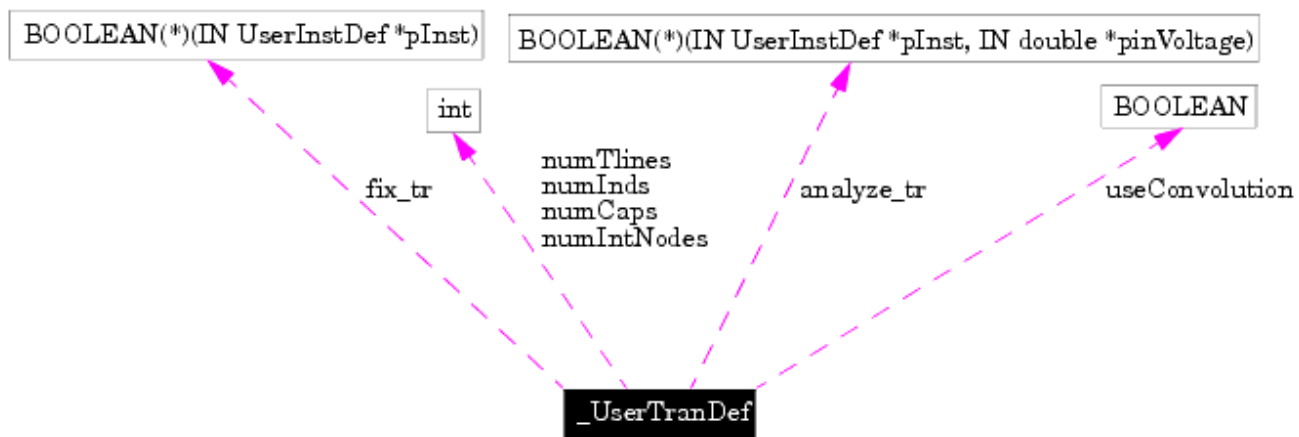
Collaboration Diagram for UserInstDef

UserNonLinDef Struct Reference



Collaboration Diagram for UserNonLinDef

UserTranDef Struct Reference



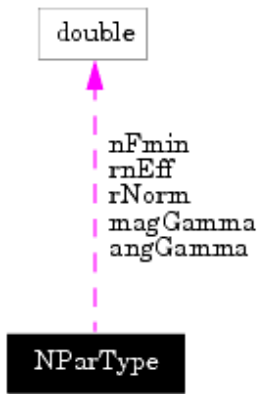
Collaboration Diagram for UserTranDef

COMPLEX Struct Reference

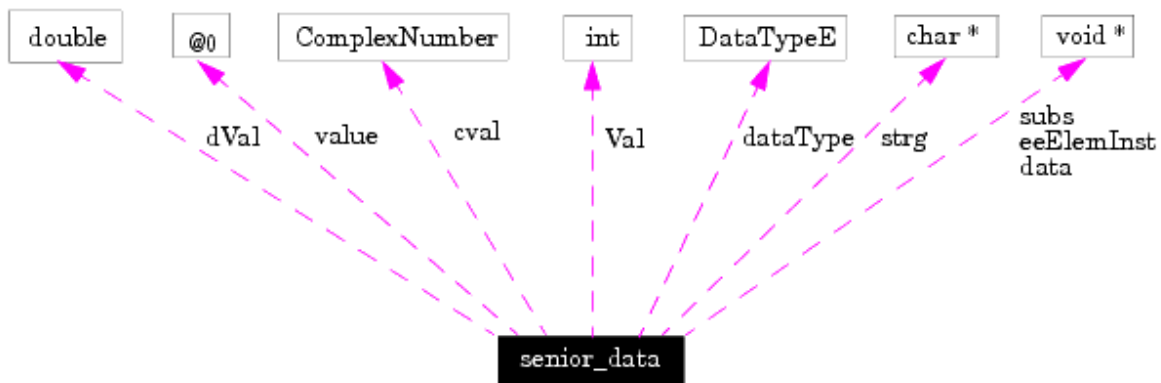


Collaboration Diagram for COMPLEX

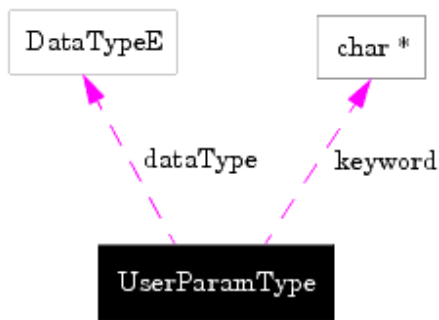
NParType Struct Reference

Collaboration Diagram for `NParType`

senior_data Struct Reference

Collaboration Diagram for `senior_data`

UserParamType Struct Reference



Collaboration Diagram for UserParamType

Typedefs

```
typedef struct senior_descriptor_data SENIOR_DESCRIPTOR_DATA
```

```
typedef struct senior_model_descriptor_data SENIOR_MODEL_DESCRIPTOR_DATA
```

```
typedef struct senior_data SENIOR_USER_DATA
```

Parameter value of an instance.

The parameter values of an item are obtained in an array of this data type.

SENIOR_USER_DATA and UserParamData are typedefs of struct senior_data.

```
typedef struct _UserElemDef SENIOR_USER_MODEL
```

```
typedef struct _UserElemDef UserElemDef
```

```
typedef struct _UserInstDef UserInstDef
```

```
typedef struct _UserNonLinDef UserNonLinDef
```

```
typedef struct _UserTranDef UserTranDef
```

Enumeration Type

enum
DataTypeE

Element parameter type.

Enumeration values:

NO_data	for parameter with unspecified data type
INT_data	for parameter with integer data type
REAL_data	for parameter with real data type
MTRL_data	for parameter referring to an instance
STRG_data	for parameter with string data type
CMPLX_data	for parameter with complex data type
INT_VECTOR_data	for parameter with integer vector data type
REAL_VECTOR_data	for parameter with real vector data type
CMPLX_VECTOR_data	for parameter with complex vector data type

Function Reference

This section provides information on User-Compiled Model application procedural interface (API) functions. The table below provides an alphabetical list of each of the functions available.

User-Defined Models

A,D	
<i>active_noise</i> (modbuild)	<i>add_tr_inductor</i> (modbuild)
<i>add_lin_n</i> (modbuild)	<i>add_tr_iq</i> (modbuild)
<i>add_lin_y</i> (modbuild)	<i>add_tr_lossy_inductor</i> (modbuild)
<i>add_nl_gc</i> (modbuild)	<i>add_tr_mutual_inductor</i> (modbuild)
<i>add_nl_iq</i> (modbuild)	<i>add_tr_resistor</i> (modbuild)
<i>add_tr_capacitor</i> (modbuild)	<i>add_tr_tline</i> (modbuild)
<i>add_tr_gc</i> (modbuild)	<i>dump_params</i> (modbuild)
E,F	
<i>ee_compute_n</i> (modbuild)	<i>ee_pre_analysis</i> (modbuild)
<i>ee_compute_y</i> (modbuild)	<i>first_frequency</i> (modbuild)
<i>ee_post_analysis</i> (modbuild)	<i>first_iteration</i> (modbuild)
G,L,M	
<i>get_delay_v</i> (modbuild)	<i>get_ucm_param_real_value</i> (modbuild)
<i>get_params</i> (modbuild)	<i>get_ucm_param_string_value</i> (modbuild)
<i>get_temperature</i> (modbuild)	<i>get_ucm_param_vector_complex_value</i> (modbuild)
<i>get_tr_time</i> (modbuild)	<i>get_ucm_param_vector_int_value</i> (modbuild)
<i>get_ucm_num_external_nodes</i> (modbuild)	<i>get_ucm_param_vector_real_value</i> (modbuild)
<i>get_ucm_num_of_params</i> (modbuild)	<i>get_ucm_param_vector_size</i> (modbuild)
<i>get_ucm_param_complex_value</i> (modbuild)	<i>get_user_inst</i> (modbuild)
<i>get_ucm_param_data_type</i> (modbuild)	<i>is_ucm_repeat_param</i> (modbuild)
<i>get_ucm_param_int_value</i> (modbuild)	<i>load_elements</i> (modbuild)
<i>get_ucm_param_name</i> (modbuild)	<i>load_elements2</i> (modbuild)
<i>get_ucm_param_num_repeats</i> (modbuild)	<i>multifile</i> (modbuild)
<i>get_ucm_param_ptr</i> (modbuild)	
P,S,V	
<i>passive_noise</i> (modbuild)	<i>send_info_to_file</i> (modbuild)
<i>print_ucm_param_value</i> (modbuild)	<i>send_info_to_scn</i> (modbuild)
<i>s_y_convert</i> (modbuild)	<i>verify_senior_parameter</i> (modbuild)
<i>send_error_to_scn</i> (modbuild)	

active_noise

```

BOOLEAN active_noise ( IN COMPLEX * yPar ,
                       IN NParType * nPar ,
                       int          numFloatPins ,
                       OUT COMPLEX * _nCor_
                       )

```



Note

For information on COMPLEX and NParType, refer to *Class List* (modbuild).

Description

This function computes the complex noise correlation 2x2 matrix for an active 3-terminal 2-port element/network, given its Y-pars and measured noise parameters.

Note that if *numFloatPins* is 2, the common (reference) third terminal is ground.

Arguments

Name	Description
<i>yPar</i>	Array of nodal admittance matrix
<i>nPar</i>	Array of noise correlation matrix
<i>numFloatPins</i>	Number of floating pins
<i>nCor</i>	An array of noise-current correlation admittance, normalized to FOUR_K_TO

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_lin_n

```

BOOLEAN add_lin_n ( INOUT UserInstDef * userInst ,
                    IN int                iPin ,
                    IN int                jPin ,
                    IN COMPLEX            iNcorr
                    )

```



Note

For information on UserInstDef and COMPLEX, refer to *Class List* (modbuild).

Description

This function adds the COMPLEX noise-current correlation term *iNcorr* (siemens, normalized to FOUR_K_TO) between (*iPin* , *jPin*). This function *must* be called from the device's analyze_ac_n() function.

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>jPin</i>	pin index
<i>iNcorr</i>	noise-current correlation admittance, normalized to FOUR_K_TO

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_lin_y

```

BOOLEAN add_lin_y ( INOUT UserInstDef * userInst ,
                    IN int                iPin ,
                    IN int                jPin ,
                    IN COMPLEX            y
                    )

```



Note

For information on UserInstDef and COMPLEX, refer to *Class List* (modbuild).

Description

This function adds the linear COMPLEX Y-parameter (*iPin* , *jPin*) branch contribution.

Note that this function can only be called from the device's analyze_lin() or analyze_ac() function even for linear capacitive branches at DC ($\omega = 0$): this will save the jacobian matrix entry location for subsequent non-zero harmonic ω .

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>jPin</i>	pin index
<i>y</i>	admittance in siemens

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_nl_gc

```

BOOLEAN add_nl_gc ( INOUT UserInstDef * userInst ,
                    IN int             iPin ,
                    IN int             jPin ,
                    IN double          g,
                    IN double          c
                    )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function adds the nonlinear conductance and capacitance contribution for the (*iPin*, *jPin*) branch: $g = d(\text{current}(iPin))/d(\text{voltage}(jPin))$ $c = d(\text{charge}(iPin))/d(\text{voltage}(jPin))$. The function *must* be called (in device's analyze_nl()).

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>jPin</i>	pin index
<i>g</i>	conductance in siemens
<i>c</i>	capacitance in farads

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_nl_iq

```

BOOLEAN add_nl_iq ( INOUT UserInstDef * userInst ,
                   IN int                iPin ,
                   IN double             current,
                   IN double             charge
                   )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function adds the nonlinear *current* and *charge* contribution at the device pin *iPin*. The function *must* be called (in device's analyze_nl()).

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>current</i>	current in amperes
<i>charge</i>	charge

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_tr_capacitor

```

BOOLEAN add_tr_capacitor ( INOUT UserInstDef * userInst ,
                           IN int                iPin ,
                           IN int                jPin ,
                           IN double             cval
                           )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function may be called (in devices's analyze_tr()) to add a capacitor between pins *iPin* and *jPin* . Values are in farads.

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>jPin</i>	pin index
<i>cval</i>	capacitance in farads

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_tr_gc

```

BOOLEAN add_tr_gc ( INOUT UserInstDef * userInst ,
                    IN int             iPin ,
                    IN int             jPin ,
                    IN double          g
                    IN double          c
                    )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function adds the transient conductance and capacitance contribution for the (*iPin*, *jPin*) branch. It *must* be called (in device's analyze_tr()).

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>jPin</i>	pin index
<i>g</i>	conductance in siemens
<i>c</i>	capacitance in farads

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_tr_inductor

```

BOOLEAN add_tr_inductor ( INOUT UserInstDef * userInst ,
                          IN int                iPin ,
                          IN int                jPin ,
                          IN double             lval
                          )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function may be called (in devices's analyze_tr()) to add a inductor between pins *iPin* and *jPin*. Values are in henrys.

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>jPin</i>	pin index
<i>lval</i>	inductance in henrys

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_tr_iq

```

BOOLEAN add_tr_iq ( INOUT UserInstDef * userInst ,
                   IN int                iPin ,
                   IN double             current
                   IN double             charge
                   )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function adds the transient current and charge contribution at the device pin *iPin*. It *must* be called (in device's analyze_tr()).

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>current</i>	current in amperes
<i>charge</i>	charge

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_tr_lossy_inductor

```

BOOLEAN add_tr_lossy_inductor ( INOUT UserInstDef * userInst ,
                                IN int                pin1 ,
                                IN int                pin2 ,
                                IN double             rval ,
                                IN double             lval
                                )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function may be called (in devices's `analyze_tr()`) to add a lossy inductor between pins *iPin* and *jPin*. Values are in henrys for inductance and in ohms for resistance.

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>jPin</i>	pin index
<i>rval</i>	resistance in ohms
<i>lval</i>	inductance in henrys

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_tr_mutual_inductor

```

BOOLEAN add_tr_mutual_inductor ( INOUT UserInstDef * userInst ,
                                IN int                ind1 ,
                                IN int                ind2 ,
                                IN double             K
                                )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function may be called (in devices's analyze_tr()) to add mutual inductance with coupling coefficient K between the inductors *ind1* and *ind2* .

Note that add_tr_inductor() or add_tr_lossy_inductor() must be added before the mutual inductance is added. *ind1* and *ind2* are the values returned from add_tr_inductor() or add_tr_lossy_inductor().

Example

```

int ind1, ind2; boolean status = TRUE; ind1 =
add_tr_lossy_inductor(userInst, 0, 2, R1, L1); ind2 =
add_tr_lossy_inductor(userInst, 1, 3, R2, L2); if( ind1 && ind2 ) status =
add_tr_mutual_inductor(userInst, ind1, ind2, K12); else status = FALSE;

```

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>ind1</i> ,	inductor index returned from add_tr_inductor or add_tr_lossy_inductor call
<i>ind2</i> ,	inductor index returned from add_tr_inductor or add_tr_lossy_inductor call
<i>K</i>	coupling coefficient, where $-1.0 < K < 1$

Returned Value

Returns TRUE if successful, FALSE otherwise.

See Also

add_tr_inductor (modbuild), *add_tr_lossy_inductor* (modbuild)

add_tr_resistor

```

BOOLEAN add_tr_resistor ( INOUT UserInstDef * userInst ,
                          IN int                iPin ,
                          IN int                jPin ,
                          IN double             rval
                          )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function may be called (in devices's analyze_tr()) to add a resistor between pins *iPin* and *jPin* . Values are in ohms.

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>jPin</i>	pin index
<i>rval</i>	resistance in ohms

Returned Value

Returns TRUE if successful, FALSE otherwise.

add_tr_tline

```

BOOLEAN add_tr_tline ( INOUT UserInstDef * userInst ,
                      IN int                pin1 ,
                      IN int                pin2 ,
                      IN int                pin3 ,
                      IN int                pin4 ,
                      IN double             z0 ,
                      IN double             td ,
                      IN double             loss
                      )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function may be called (in devices's fix_tr()) to add an ideal transmission line.

The inputs are *pin1* (positive) and *pin3* (negative), the outputs are *pin2* (positive) and *pin4* (negative), the impedance is *z0* , in ohms, and the delay time is *td* , in seconds. the loss is an attenuation scale factor; a lossless line has loss=1.0

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>pin1</i>	pin index of the input positive pin
<i>pin2</i>	pin index of the output positive pin
<i>pin3</i>	pin index of the input negative pin
<i>pin4</i>	pin index of the output negative pin
<i>z0</i>	impedance in ohms
<i>td</i>	delay time in seconds
<i>loss</i>	attenuation scale factor

Returned Value

Returns TRUE if successful, FALSE otherwise.

dump_params

BOOLEAN dump_params (IN void * eeElemInst)

Description

This function prints out the instance *eeElemInst* 's parameter names and values to stderr.

This function should only be used for debugging purposes.

Arguments

Name	Description
<i>eeElemInst</i>	ADS element instance. For example, if the ADS model instance named "Msub1" is defined as the first parameter of the userCompiled model, *eeElemInst is userInst->pData[0].value.eeElemInst

Returned Value

Returns TRUE if successful, FALSE otherwise.

ee_compute_n

```

BOOLEAN ee_compute_n ( INOUT void *      eeElemInst ,
                       IN UserParamData * pData ,
                       IN double          omega ,
                       IN COMPLEX *      yPar ,
                       OUT COMPLEX *     nCor ,
                       )

```



Note

For information on UserParamData and COMPLEX, refer to *Class List* (modbuild).

Description

This function allows access to ADS elements for noise analysis.

Note that parameter data *pData* must be supplied in SI units, where applicable.

Arguments

Name	Description
<i>eeElemInst</i>	ADS element instance. For example, if the ADS model instance name "Msub1" is defined as the first parameter of the userCompiled model, *eeElemInst is userInst->pData[0].value.eeElemInst
<i>pData</i>	instance's parameters
<i>omega</i>	frequency omega radians/sec
<i>yPar</i>	Array of nodal admittance matrix
<i>nCor</i>	An array of noise-current correlation admittance, normalized to FOUR_K_TO

Returned Value

Returns TRUE if successful, FALSE otherwise.

ee_compute_y

```

BOOLEAN ee_compute_y ( INOUT void *      eeElemInst ,
                       IN UserParamData * pData ,
                       IN double          omega ,
                       OUT COMPLEX *      yPar
                       )

```



Note

For information on UserParamData and COMPLEX, refer to *Class List* (modbuild).

Description

This function allows access to ADS elements for linear analysis.

Note that parameter data *pData* must be supplied in SI units, where applicable.

Arguments

Name	Description
<i>eeElemInst</i>	ADS element instance. For example, if the ADS model instance name "Msub1" is defined as the first parameter of the userCompiled model, *eeElemInst is userInst->pData[0].value.eeElemInst
<i>pData</i>	instance's parameters
<i>omega</i>	frequency omega radians/sec
<i>yPar</i>	Array of nodal admittance matrix

Returned Value

Returns TRUE if successful, FALSE otherwise.

ee_post_analysis

BOOLEAN ee_post_analysis (INOUT void * eeElemInst)

Description

This function *must* be called by User-Defined Model code (possibly from a User-Defined Model element's post_analysis() function) for every ee_pre_analysis() call to free memory allocated for the ADS instance *eeElemInst* .

Arguments

Name	Description
<i>eeElemInst</i>	ADS element instance. For example, if the ADS model instance named "Msub1" is defined as the first parameter of the userCompiled model, *eeElemInst is userInst->pData[0].value.eeElemInst
<i>yPar</i>	Array of nodal admittance matrix

Returned Value

Returns TRUE if successful, FALSE otherwise.

ee_pre_analysis

```
BOOLEAN ee_pre_analysis ( IN char *          eName ,
                          IN UserParamData * pData ,
                          )
```



Note

For information on UserParamData, refer to *Class List* (modbuild).

Description

This function *must* be called by User-Defined Model code which require that you use an ADS element (possibly from a User-Defined Model element's `pre_analysis()` function).

The function returns a pointer to an allocated ADS element instance if successful, NULL otherwise. This pointer must be saved (possibly with the User-Defined Model element instance, in its *seniorData* field) and passed to `ee_compute_y()` or `ee_compute_n()`.

Arguments

Name	Description
<i>eName</i>	element name
<i>pData</i>	instance's parameters

Returned Value

Returns a pointer to an allocated ADS element instance if successful, NULL otherwise.

See Also

`ee_compute_n` (modbuild), `ee_compute_y` (modbuild)

first_frequency

BOOLEAN first_frequency(void)

first_iteration

BOOLEAN first_iteration(void)

get_ucm_num_external_nodes

```
int get_ucm_num_external_nodes( const UserInstDef* userInst )
```

**Note**

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function returns the number of external nodes.

Arguments

Name	Description
<i>userInst</i>	user element instance

Returned Value

Returns the number of external nodes.

get_ucm_num_of_params

```
int get_ucm_num_of_params( const UserInstDef* userInst )
```

**Note**

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function returns number of parameters in the parameter definition. No matter how many entries a repeated parameter has, it is counted as one parameter.

Arguments

Name	Description
<i>userInst</i>	user element instance

Returned Value

Returns the number of parameters in the parameter definition.

get_ucm_param_complex_value

Complex get_ucm_param_complex_value (const SENIOR_USER_DATA *param, BOOLEAN* status, char* errorMsg)

The function returns parameter value which is type of complex. It can be used only when the parameter value type is complex. The returned value is valid only when *status* is 1. If the query fails, *status* is set to zero and error message is written to *errorMsg*.

Arguments

Name	Description
<i>param</i>	the pointer to a parameter, obtained from get_ucm_param_ptr() function.
<i>status</i>	query status. It is set to 1 if no error occurs, otherwise, it is set to 0.
<i>errorMsg</i>	error message when query fails. Enough buffer should be allocated for it, for example, 2048.

Returned Value

Returns the parameter complex value.

get_ucm_param_data_type

DataTypeE get_ucm_param_data_type(const SENIOR_USER_DATA* param)

**Note**

For information on SENIOR_USER_DATA and DataTypeE, refer to *Class List* (modbuild).

Description

This function returns the data type of the parameter value.

Arguments

Name	Description
<i>param</i>	the pointer to a parameter, obtained from get_ucm_param_ptr() function.

Returned Value

Returns the data type of the parameter value.

get_ucm_param_int_value

int get_ucm_param_int_value (const SENIOR_USER_DATA *param, BOOLEAN* status, char* errorMsg)

The function returns parameter value which is type of integer. It can be used only when the parameter value type is integer. The returned value is valid only when *status* is 1. If the query fails, *status* is set to zero and error message is written to *errorMsg*.

Arguments

Name	Description
<i>param</i>	the pointer to a parameter, obtained from get_ucm_param_ptr() function.
<i>status</i>	query status. It is set to 1 if no error occurs, otherwise, it is set to 0.
<i>errorMsg</i>	error message when query fails. Enough buffer should be allocated for it, for example, 2048.

Returned Value

Returns the parameter integer value.

get_ucm_param_name

```
const char* get_ucm_param_name( const UserInstDef* userInst, int paramIndex )
```

**Note**

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function returns the parameter name.

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>paramIndex</i>	the parameter index, starting from 0

Returned Value

Returns the parameter name.

get_ucm_param_num_repeats

```
int get_ucm_param_num_repeats( const UserInstDef* userInst, int paramIndex )
```

**Note**

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function returns the number of entries of a repeated parameter. For example, Freq[1] and Freq[2] are specified, then the number of entries is 2.

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>paramIndex</i>	the parameter index, starting from 0

Returned Value

Returns the number of entries of a repeated parameter.

get_ucm_param_ptr

```
const SENIOR_USER_DATA* get_ucm_param_ptr( const UserInstDef* userInst, int
paramIndex, char* errorMsg, int repeatIndex )
```



Note

For information on UserInstDef and SENIOR_USER_DATA, refer to *Class List* (modbuild).

Description

This function returns the pointer to a parameter. Before querying a parameter value, this function should be used to obtain the pointer to the parameter. NULL pointer is returned if the parameter not found or an error occurs. The error message is stored in errorMsg. Please allocate enough buffer size for errorMsg.

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>paramIndex</i>	the parameter index, starting from 0
<i>errorMsg</i>	error message
<i>repeatIndex</i>	the repeat index of a repeated parameter, starting from 0. For example, Freq[2], repeatIndex=1. For non-repeated parameter, use -1.

Returned Value

Returns the pointer to a parameter. NULL if the parameter is not found or an error occurs.

get_ucm_param_real_value

double get_ucm_param_real_value (const SENIOR_USER_DATA *param, BOOLEAN* status, char* errorMsg)

Description

The function returns parameter value which is type of real. It can be used only when the parameter value type is real. The returned value is valid only when *status* is 1. If the query fails, *status* is set to zero and error message is written to *errorMsg*.

Arguments

Name	Description
<i>param</i>	the pointer to a parameter, obtained from get_ucm_param_ptr() function.
<i>status</i>	query status. It is set to 1 if no error occurs, otherwise, it is set to 0.
<i>errorMsg</i>	error message when query fails. Enough buffer should be allocated for it, for example, 2048.

Returned Value

Returns the parameter real value.

get_ucm_param_string_value

```
const char* get_ucm_param_string_value (const SENIOR_USER_DATA *param,  
BOOLEAN* status, char* errorMsg)
```

The function returns parameter value which is type of string. It can be used only when the parameter value type is string. The returned value is valid only when *status* is 1. If the query fails, *status* is set to zero and error message is written to *errorMsg*.

Arguments

Name	Description
<i>param</i>	the pointer to a parameter, obtained from get_ucm_param_ptr() function.
<i>status</i>	query status. It is set to 1 if no error occurs, otherwise, it is set to 0.
<i>errorMsg</i>	error message when query fails. Enough buffer should be allocated for it, for example, 2048.

Returned Value

Returns the parameter string value.

get_ucm_param_vector_complex_value

Complex get_ucm_param_vector_complex_value (const SENIOR_USER_DATA *param, int index, BOOLEAN* status, char* errorMsg)

Description

The function returns parameter complex value. It can be used only when the parameter has vector value and the value type is complex. The returned value is valid only when *status* is 1. If the query fails, *status* is set to zero and error message is written to *errorMsg*.

Arguments

Name	Description
<i>param</i>	the pointer to a parameter, obtained from get_ucm_param_ptr() function.
<i>index</i>	index in the array, starting from 0.
<i>status</i>	query status. It is set to 1 if no error occurs, otherwise, it is set to 0.
<i>errorMsg</i>	error message when query fails. Enough buffer should be allocated for it, for example, 2048.

Returned Value

Returns the complex value of the *index_{th}* item from the parameter's complex vector value list.

get_ucm_param_vector_int_value

int get_ucm_param_vector_int_value (const SENIOR_USER_DATA *param, int index, BOOLEAN* status, char* errorMsg)

Description

The function returns parameter integer value. It can be used only when the parameter has vector value and the value type is integer. The returned value is valid only when *status* is 1. If the query fails, *status* is set to zero and error message is written to *errorMsg*.

Arguments

Name	Description
<i>param</i>	the pointer to a parameter, obtained from get_ucm_param_ptr() function.
<i>index</i>	index in the array, starting from 0.
<i>status</i>	query status. It is set to 1 if no error occurs, otherwise, it is set to 0.
<i>errorMsg</i>	error message when query fails. Enough buffer should be allocated for it, for example, 2048.

Returned Value

Returns the integer value of the *index_th* item from the parameter's integer vector value list.

get_ucm_param_vector_real_value

double get_ucm_param_vector_real_value (const SENIOR_USER_DATA *param, int index, BOOLEAN* status, char* errorMsg)

Description

The function returns parameter real value. It can be used only when the parameter has vector value and the value type is real. The returned value is valid only when *status* is 1. If the query fails, *status* is set to zero and error message is written to *errorMsg*.

Arguments

Name	Description
<i>param</i>	the pointer to a parameter, obtained from get_ucm_param_ptr() function.
<i>index</i>	index in the array, starting from 0.
<i>status</i>	query status. It is set to 1 if no error occurs, otherwise, it is set to 0.
<i>errorMsg</i>	error message when query fails. Enough buffer should be allocated for it, for example, 2048.

Returned Value

Returns the real value of the *index_th* item from the parameter's real vector value list.

get_ucm_param_vector_size

```
int get_ucm_param_vector_size( const SENIOR_USER_DATA* param )
```

**Note**

For information on SENIOR_USER_DATA, refer to *Class List* (modbuild).

Description

This function returns the vector size for a parameter which has vector value.

Arguments

Name	Description
<i>param</i>	the pointer to a parameter, obtained from get_ucm_param_ptr() function.

Returned Value

Returns the vector size if the parameter has vector value, otherwise 0.

get_delay_v

```

BOOLEAN get_delay_v ( INOUT UserInstDef * userInst ,
                      IN int                iPin ,
                      IN int                jPin ,
                      IN double             tau
                      OUT double *          vDelay
                      )

```



Note

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function may be called (from device's `analyze_ni()`) to get *tau* seconds delay * (*iPin* , *jPin*) voltage difference.

GND may be used as *jPin* to get absolute (w.r.t. ground) delayed pin voltage.

Note that *tau* must not be dependent device pin voltages (i.e. it is an ideal delay).

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>iPin</i>	pin index
<i>jPin</i>	pin index
<i>tau</i>	delay in seconds
<i>vDelay</i>	array of delayed pin voltage

Returned Value

Returns TRUE if successful, FALSE otherwise.

get_params

```

BOOLEAN get_parms ( IN void *          eeElemInst ,
                    OUT UserParamData * pData ,
                    )

```



Note

For information on UserParamData, refer to *Class List* (modbuild).

Description

This function loads the passed instance *eeElemInst* 's RHS parameter values into *pData* , which must be big enough to store all parameters.

It is mainly useful to obtain a referred instance's (such as MSUB, TEMP) parameters. Note that a User-Defined Model instance's parameters are always available in the *UserInstDef.pData* array, so there is no need to call this for a user-instance's own params.

Arguments

Name	Description
<i>eeElemInst</i>	ADS element instance. For example, if the ADS model instance name "Msub1" is defined as the first parameter of the userCompiled model, *eeElemInst is userInst->pData[0].value.eeElemInst
<i>pData</i>	instance's parameters

Returned Value

Returns TRUE if successful, FALSE otherwise.

get_temperature

BOOLEAN get_temperature(void)

Description

This function returns the value of the ADS global variable *temp* .

Returned Value

Returns the value of temperature in kelvin.

get_tr_time

BOOLEAN get_tr_time(void)

Description

This function retrieves the current transient analysis time in seconds.

Returned Value

Returns the current transient analysis time in seconds.

get_user_inst

UserInstDef* get_user_inst(IN void * *eeElemInst*)

Description

This returns a pointer to the *UserInstDef* User-Defined Model instance if *eeElemInst* is indeed an instance of a User-Defined Model element, NULL otherwise.

Arguments

Name	Description
<i>eeElemInst</i>	ADS element instance. For example, if the ADS model instance name "Msub1" is defined as the first parameter of the userCompiled model, * <i>eeElemInst</i> is userInst->pData[0].value. <i>eeElemInst</i>

Returned Value

Returns a pointer to the *UserInstDef* User-Defined Model instance if *eeElemInst* is an instance of a User-Defined Model element, NULL otherwise.

is_ucm_repeat_param

BOOLEAN is_ucm_repeat_param(const UserInstDef* userInst, int paramIndex)

**Note**

For information on UserInstDef, refer to *Class List* (modbuild).

Description

This function returns whether the parameter is repeated parameter.

Arguments

Name	Description
<i>userInst</i>	user element instance
<i>paramIndex</i>	the parameter index, starting from 0

Returned Value

Returns TRUE if the parameter is repeated parameter, otherwise FALSE.

load_elements

```
BOOLEAN load_elements ( IN UserElemDef * userElem ,
                        IN int          numElem
                        )
```



Note

For information on UserElemDef, refer to *Class List* (modbuild).

Description

This function is used to load a single User-Defined Model module.

Note that this function should be used if versions and (*modify_param) are not defined as members of the struct *UserElemDef* , otherwise load_elements2() should be used.

Arguments

Name	Description
<i>userElem</i>	An array of UserElemDef which contains all user-defined elements in the module
<i>numElem</i>	The size of UserElemDef array which is also the number of use-defined elements in the module

Returned Value

Returns TRUE if successful, FALSE otherwise.

See Also

load_elements2 (modbuild)

load_elements2

```
BOOLEAN load_elements2 ( IN UserElemDef * userElem,
                        IN int          numElem
                        )
```



Note

For information on UserElemDef, refer to *Class List* (modbuild).

Description

This function is used to load a single User-Defined Model module. The function was introduced after ADS 2003C.

Note that this function should be used if versions and (*modify_param) are defined as members of the struct *UserElemDef* , otherwise load_elements() should be used.

Arguments

Name	Description
<i>userElem</i>	An array of UserElemDef which contains all user-defined elements in the module
<i>numElem</i>	The size of UserElemDef array which is also the number of use-defined elements in the module

Returned Value

Returns TRUE if successful, FALSE otherwise.

See Also

load_elements (modbuild)

multifile

BOOLEAN multifile(void)

Description

ADS entry function to boot all User-Defined Model modules, defined in *userindx.c*. Each User-Defined Model module must have a call to its booting function here.

It is never modified by the Advanced Design System's Model Builder interface.

This module enables the user to link multiple User-Defined Model modules with the main program. It works as an *intermediary* between the main program and all of the User-Defined Model modules: it calls the boot function (use the name "boot_abc()" for the module named "abc.c") to load the element/data item definitions in that module. The boot_abc() function must in turn call load_elements() to accomplish this.

Note the example calls below:

boot_userex() to boot module "userex.c", boot_mymodule() to boot module "mymodule.c"

If you want to ignore booting failures, ensure that multifile() always returns TRUE and remove the extra return statements. Also, remove the send_error_to_scn() calls to reduce excess status messages during run time.

passive_noise

```

BOOLEAN passive_noise ( IN COMPLEX * yPar ,
                        IN double      tempC ,
                        IN int         numNodes
                        ,
                        OUT COMPLEX * nCor
                        )

```



Note

For more information on COMPLEX, refer to *Class List* (modbuild).

Description

This function computes the complex noise correlation matrix for a passive element, given its Y-pars, operating temperature (Celsius) and # of nodes.

Arguments

Name	Description
<i>yPar</i>	Array of nodal admittance matrix
<i>tempC</i>	Temperature in Celsius
<i>numNodes</i>	Number of nodes
<i>nCor</i>	Array of noise-current correlation admittance, normalized to FOUR_K_TO

Returned Value

Returns TRUE if successful, FALSE otherwise.

print_ucm_param_value

```
void print_ucm_param_value( FILE *fp, const SENIOR_USER_DATA *param )
```

**Note**

For information on SENIOR_USER_DATA, refer to *Class List* (modbuild).

Description

This function prints a parameter value and data type to output specified by the first argument **fp*. It is for debug usage. Its implementation is listed in `$HPEESOF_INSTALLATION_DIR/modelbuilder/lib/printParam.c` as sample code on how to query parameter values.

Arguments

Name	Description
<i>fp</i>	pointer to a output file
<i>param</i>	pointer to the parameter, obtained from <code>get_ucm_param_ptr()</code> function

Returned Value

None.

send_error_to_scn

```
void send_error_to_scn( IN char * msg )
```

Description

This function is useful to indicate program status in various stages of execution. For example, during module bootup, element analyses, and pre- or post-analysis.

The function writes *msg* to the Errors/Warnings window.

Arguments

Name	Description
<i>msg</i>	The string to be sent to the simulation Errors/Warning window

send_info_to_file

```
void send_info_to_file( IN char * msg )
```

Description

This function is useful to indicate program status in various stages of execution. For example, during module bootup, element analyses, and pre- or post-analysis.

The function writes *msg* to the *.inf* file.

Arguments

Name	Description
<i>msg</i>	The string to be written to the <i>.inf</i> file

send_info_to_scn

```
void send_info_to_scn( IN char * msg )
```

Description

This function is useful to indicate program status in various stages of execution. For example, during module bootup, element analyses, and pre- or post-analysis.

The function writes *msg* to the simulation Status/Progress window.

Arguments

Name	Description
<i>msg</i>	The string to be sent to the simulation Status/Progress window

s_y_convert

```

BOOLEAN s_y_convert ( IN COMPLEX * inPar ,
                      OUT COMPLEX * outPar,
                      IN int         direction
                      ,
                      IN double      rNorm ,
                      IN int         size
                      )

```



Note

For more information on COMPLEX, refer to *Class List* (modbuild).

Description

This function converts between S- and Y- parameters.

Arguments

Name	Description
<i>inPar</i>	Input parameter array
<i>outPar</i>	Output parameter array
<i>direction</i>	0: inPar(Y-pars) -> outPar(S-pars) 1: inPar(S-pars) -> outPar(Y-pars)
<i>rNorm</i>	The S-parameter normalizing impedance in ohms
<i>size</i>	The matrix size

Returned Value

Returns TRUE if successful, FALSE otherwise.

verify_senior_parameter

```
ee_bool          ( SENIOR_USER_DATA * data ,
verify_senior_parameter

                  DataTypeE          type
                  )
```



Note

For more information on SENIOR_USER_DATA, refer to *Class List* (modbuild).

Defines

```
#define BOLTZ 1.380658e-23
    use NIST Codata-86 value
#define boolean int
#define BOOLEAN int
#define CHARGE 1.60217733e-19
    use NIST Codata-86 value
#define ComplexNumber COMPLEX
#define CTOK 273.15
#define FALSE 0
#define FOUR_K_TO (4.0*BOLTZ*NOISE_REF_TEMP)
    noise normalization 4kToB, B=1 Hz
#define get_angunit ( eeElemInst ) 1.0
    angle scale factor in the scope of eeElemInst. It is defined to keep ADS compatibility with SIV design units
#define get_cunit ( eeElemInst ) 1.0
    capacitance scale factor in the scope of eeElemInst. It is defined to keep ADS compatibility with SIV
    design units
#define get_curunit ( eeElemInst ) 1.0
    current scale factor in the scope of eeElemInst. It is defined to keep ADS compatibility with SIV design
    units
#define get_funit ( eeElemInst ) 1.0
    frequency scale factor in the scope of eeElemInst. It is defined to keep ADS compatibility with SIV design
    units
#define get_gunit ( eeElemInst ) 1.0
    conductance scale factor in the scope of eeElemInst. It is defined to keep ADS compatibility with SIV
    design units
#define get_lenunit ( eeElemInst ) 1.0
    length scale factor in the scope of eeElemInst. It is defined to keep ADS compatibility with SIV design units
#define get_lunit ( eeElemInst ) 1.0
    inductance scale factor in the scope of eeElemInst. It is defined to keep ADS compatibility with SIV design
    units
#define get_runit ( eeElemInst ) 1.0
    resistance scale factor in the scope of eeElemInst. It is defined to keep ADS compatibility with SIV design
    units
#define get_tunit ( eeElemInst ) 1.0
```

User-Defined Models

time scale factor in the scope of eeElemInst. It is defined to keep ADS compatibility with SIV design units

```
#define get_volunit ( eeElemInst ) 1.0
```

voltage scale factor in the scope of eeElemInst. It is defined to keep ADS compatibility with SIV design units

```
#define get_watt ( eeElemInst, power ) (power)
```

return watt-equivalent of 'power' in eeElemInst's scope. It is defined to keep ADS compatibility with SIV design units

```
#define GND -1
```



```
#define IN
```

input argument to function

```
#define INOUT
```

argument used and modified by function

```
#define LOCAL static
```



```
#define NOISE_REF_TEMP 290.0
```

standard noise reference temperature, in Kelvin

```
#define NULL 0L
```

The following macros are defined in Libra and are copied here.

```
#define RealNumber double
```



```
#define size ( thing ) (sizeof(thing)/sizeof(*thing))
```



```
#define TRUE 1
```



```
#define TWOPI 6.28318530717958623199593
```

define 2 * PI

Also included are Senior user-needed interface function declarations. Warning: Do not modify any existing definition/declaration here. This file provides the definitions for the Libra-Senior user defined models.

```
#define UNUSED
```

unused argument

```
#define UserParamData SENIOR_USER_DATA
```

UserParamData A macro for struct SENIOR_USER_DATA.