User-Defined Models

December 2003
Notice

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

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

Warranty

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

Restricted Rights Legend

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

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


Acknowledgments

Mentor Graphics is a trademark of Mentor Graphics Corporation in the U.S. and other countries.

Microsoft®, Windows®, MS Windows®, Windows NT®, and MS-DOS® 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.

Java™ is a U.S. trademark of Sun Microsystems, Inc.
## Contents

1 Building User-Compiled Analog Models

<table>
<thead>
<tr>
<th>Background</th>
<th>Using the Model Development Kit</th>
<th>Starting a New User-Compiled Model</th>
<th>Identifying the Circuit Type</th>
<th>Using the User-Compiled Circuit Model Dialog</th>
<th>Specifying the Model Parameters</th>
<th>Compiling the Circuit Model</th>
<th>Linking Model Files to the Simulator</th>
<th>Example - Creating a New User-Compiled Analog Model</th>
<th>Characteristics of User-Compiled Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>1-3</td>
<td>1-3</td>
<td>1-3</td>
<td>1-4</td>
<td>1-5</td>
<td>1-7</td>
<td>1-14</td>
<td>1-16</td>
<td>1-28</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Open File Dialog</th>
<th>Save As Dialog</th>
<th>Code Options (Analog/RF Models) Dialog</th>
<th>Compile Options Dialog</th>
<th>Link Options</th>
<th>Message: Confirm Files Out-of-Synch</th>
<th>Starting a User-Compiled Model</th>
<th>Defining the Model Parameters</th>
<th>Creating the Model Symbol</th>
<th>Setting Options</th>
<th>Creating the Code and Compiling the Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-9</td>
<td>1-10</td>
<td>1-10</td>
<td>1-11</td>
<td>1-15</td>
<td>1-15</td>
<td>1-16</td>
<td>1-17</td>
<td>1-20</td>
<td>1-21</td>
<td>1-22</td>
</tr>
</tbody>
</table>

2 Porting Libra Senior to the ADS 1.0 Model Builder Interface

<table>
<thead>
<tr>
<th>Data Items</th>
<th>Default Units</th>
<th>Substrates/Built-In Models</th>
<th>AEL Changes</th>
<th>Managing Model Files</th>
<th>Building the User-Compiled Model from the Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-47</td>
<td>1-48</td>
<td>1-48</td>
<td>1-49</td>
<td>1-50</td>
<td>1-51</td>
</tr>
</tbody>
</table>
## Table of Contents

1. Accessing Dynamically Loaded Devices .......................................................... 1-54
2. Creating Linear Circuit Elements
   - Deriving S-Parameter Equations ...................................................................... 2-1
   - Deriving Y-Parameter Equations ..................................................................... 2-4
   - Coding a Linear Element .................................................................................. 2-6
     - Pi-Section Resistive Attenuator ..................................................................... 2-9
   - Transmission Line Section ................................................................................ 2-15
     - Deriving an S-Parameter ............................................................................... 2-15
     - Separating the Expressions .......................................................................... 2-16
   - Algorithms........................................................................................................ 2-18
     - Applying a Problem to the Coaxial Cable Section ......................................... 2-19
     - Calculating Remaining Expressions ................................................................ 2-22
     - Adding Noise Characteristics ........................................................................ 2-23
3. Creating Nonlinear Circuit Elements
   - Requirements for Creating Nonlinear Elements ................................................. 3-1
     - Linear Part ..................................................................................................... 3-2
     - Nonlinear Part ............................................................................................... 3-2
     - AC Part .......................................................................................................... 3-3
   - User-defined P-N Diode Model .......................................................................... 3-4
     - Defining a Nonlinear Element ........................................................................ 3-5
     - Referencing Data Items .................................................................................. 3-17
     - Displaying Error/Warning Messages ............................................................... 3-17
4. Creating Transient Circuit Elements
   - Requirements for Creating Transient Elements ............................................... 4-1
     - Using Resistors, Capacitors, and Inductors .................................................... 4-2
     - Using Transmission Lines ............................................................................... 4-3
   - User-defined P-N Diode Model .......................................................................... 4-4
     - Defining the Transient Device ........................................................................ 4-5
     - Transient Analysis Function .......................................................................... 4-6
     - Referencing Data Items .................................................................................. 4-7
     - Displaying Error/Warning Messages ............................................................... 4-7
5. Custom Modeling with Symbolically-Defined Devices
   - Writing SDD Equations ..................................................................................... 5-3
     - Port Variables ................................................................................................. 5-3
     - Defining Constitutive Relationships with Equations ...................................... 5-4
     - Explicit Representation ................................................................................... 5-4
     - Implicit Representation ................................................................................... 5-5
     - Explicit Versus Implicit Representations ......................................................... 5-6
     - Continuity ........................................................................................................ 5-7
Chapter 1: Building User-Compiled Analog Models

The Model Development Kit provides a graphical user interface that enables you to create your own circuit element models. Creating a model consists of three main steps:

- Defining the parameters whose values will be entered from the schematic
- Defining the symbol and the number of pins
- Writing the C-code itself

When appropriately coded, these elements can be used in linear, nonlinear (Harmonic Balance), transient, and Circuit Envelope simulations. The ADS Application Extension Language (AEL) provides the coupling between the parameters in the schematic design environment and the simulator. The Model Development Kit manages the AEL code generation completely. It also provides a template of the C code so all you need to do is add the relevant equations. Pin numbers and parameters are provided in easy-to-remember C macros and are automatically determined from the symbol drawing.

Note: To use this tool, you must have the appropriate C++ compiler installed on your computer. For details, refer to the installation manual for your platform.
Building User-Compiled Analog Models

Background

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 an executable program. This executable program serves as a replacement for the default circuit simulator program. 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.

You supply the following information using ANSI-C code and AEL (either written by you or generated by the Model Development Kit interface):

- Definition (element names, number of pins, parameter list)
- Modeling functions (linear/nonlinear/transient/noise)
- Library and palette definitions

Note that although the user model is written in C, a C++ compiler is necessary for linking the entire program. The newly integrated elements can then be used in the same way as the built-in elements. A complete model can be developed, compiled and used in a simulation without exiting the schematic design environment.

Note: This chapter shows an example on the PC and uses the back-slash (\) convention for file pathnames. On UNIX systems, the pathnames use the (forward) slash (/).
Using the Model Development Kit

The following topics describe how to use the Advanced Design System's Model Development Kit. Before using the Model Development Kit, verify that Analog/RF is the current design type. The kit enables you to complete the following tasks:

- Starting a new model
- Identifying the circuit type
- Defining the model parameters
- Creating a model symbol and adding pins
- Writing and compiling a model
- Setting options
- Linking a model to the simulator

Starting a New User-Compiled Model

To begin building a new model:

1. Open a project.
2. From the Schematic window, choose **Tools > User-Compiled Model > New User-Compiled Model**.

The resulting dialog box enables you to provide a filename for your new user-compiled model. You must use a name that does not already exist in the current project and is not the name of a primitive used by the simulator. Type the desired name in the **Model Name** box, then click **OK**. The Circuit Type dialog box appears next.

Identifying the Circuit Type

The Circuit Type dialog box enables you to specify whether your new Analog/RF user-defined model will be linear or nonlinear. Click on the appropriate choice, then click **OK**. The User-Compiled Circuit Model dialog box appears next.

- **Linear** can have linear elements only
- **Nonlinear** can have both linear and nonlinear elements
Using the User-Compiled Circuit Model Dialog

The User-Compiled Circuit Model dialog contains the following tabs enabling you to complete the listed actions:

<table>
<thead>
<tr>
<th>Tab</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>- Define input parameters. See “Specifying the Model Parameters” on page 1-5.</td>
</tr>
</tbody>
</table>
| Model Code | - Create/edit the symbol and its pins  
|          |   - Copy a template for use with new or existing code  
|          |   - Open an existing C code file and edit it  
|          |   - Define various simulation functions  
|          |   - Set compiling options  
|          |   - View compilation status  
|          |   - Save the C code file  
|          | See “Compiling the Circuit Model” on page 1-7. |
| Options  | - Supply a description for the model  
|          |   - Change the default instance name  
|          |   - Specify which library the model should be stored in  
|          |   - Choose whether or not to include this model in Bill of Materials  
|          |   - Specify whether or not to use an external text editor (and which one)  
|          |   - Specify layout artwork information  
|          | See “Specifying the Model Options” on page 1-12. |

Message: Confirm Model File Not Created

This message appears when you attempt to dismiss this dialog box without having compiled a model. Click Yes if you wish to compile a model at this point. Otherwise, click No.
Specifying the Model Parameters

The Parameters tab enables you to define parameters and other specifications for an Analog/RF user-defined model.

Table 1-1. User-Compiled Circuit Model Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Name</td>
<td>Displays the previously assigned name of the selected user-defined model.</td>
</tr>
<tr>
<td>Select Parameter</td>
<td>Lists all of the parameters defined for this model.</td>
</tr>
<tr>
<td>Add</td>
<td>Allows you to add a newly defined parameter to the Select Parameter list after specifications are made in the fields listed under Edit Parameter.</td>
</tr>
<tr>
<td>Cut</td>
<td>Removes a selected parameter from the Select Parameter list.</td>
</tr>
<tr>
<td>Paste</td>
<td>Inserts the last parameter that was cut from the Select Parameter list.</td>
</tr>
<tr>
<td>Add Multiplicity Factor (_M)</td>
<td>Add a multiplicity factor to the model.</td>
</tr>
<tr>
<td>Copy Parameters From</td>
<td>Choose this button to bring up the Library List dialog box to choose a component. When you select OK, the selected component's parameters will be copied to the parameter list and the associated design (.dsn) file will be copied and saved as &lt;component_name&gt;.dsn in the networks directory of the current project. The Layout Artwork Type and Name are also copied.</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Type in an appropriate parameter name.</td>
</tr>
<tr>
<td>Value Type</td>
<td>Allows you to select a real (default), integer, complex, or string value type.</td>
</tr>
<tr>
<td>Default Value (e.g., 1.23e-12)</td>
<td>Specify the value to be assigned to each parameter when the component is first inserted.</td>
</tr>
<tr>
<td>Parameter Type</td>
<td>(Optional) From the drop-down list, select the type of scale factors to be assigned to the entered value (for example, Ohms, KOhms, MOhms, etc. for resistance). Default is Unitless.</td>
</tr>
<tr>
<td>Parameter Description</td>
<td>(Optional) Type in a brief description of the parameter. Default is blank.</td>
</tr>
<tr>
<td>Display parameter on schematic</td>
<td>If you check this box, the parameter name and value will be displayed on the schematic.</td>
</tr>
<tr>
<td>Optimizable</td>
<td>If you check this box, the parameter can be optimized.</td>
</tr>
</tbody>
</table>
Building User-Compiled Analog Models

Table 1-1. User-Compiled Circuit Model Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow statistical distribution</td>
<td>If you check this box, the parameter can be used in yield analysis.</td>
</tr>
<tr>
<td>OK</td>
<td>Choose this button when you have made all desired specifications in the tabs of the User-Compiled Circuit Model dialog box. The component information will be written to a file called <code>&lt;component_name&gt;.ael</code> and saved to the database. The program checks for a <code>&lt;component_name&gt;.c</code> file and will prompt you to generate one if such a file is not found. The dialog box is dismissed.</td>
</tr>
<tr>
<td>Apply</td>
<td>Choose this button when you have entered all specifications for a particular parameter. The template file will be regenerated to update the parameter data. The component information will be written to a file called <code>&lt;component_name&gt;.ael</code> and saved to the database.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Choose this button to cancel any parameter data you have changed in the Parameters Tab. This will cancel all changes since the previous save, dismiss the User-Compiled Circuit Model dialog box. Also, if the button “Create/Edit Symbols and Pins” was selected in the Model Code tab, the Schematic window that was opened in symbol view will now close.</td>
</tr>
</tbody>
</table>
Compiling the Circuit Model

The Model Code tab enables you to generate, modify, and compile your Analog/RF user-defined model.

Table 1-2. User-Compiled Circuit Model Code Generation Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Edit Symbol and Pins</td>
<td>Opens a Schematic window in the Symbol View mode. You can create or edit a symbol. The symbol is loaded if it exists. If the &lt;component&gt;.dsn file is not found, the SYM_CUI.dsn file is copied to the current project's network directory as &lt;component&gt;.dsn and loaded.</td>
</tr>
<tr>
<td>Code Options</td>
<td>Brings up the Code Options dialog box for setting specifications for the code, such as type of analysis function. See “Code Options (Analog/RF Models) Dialog” on page 1-10.</td>
</tr>
<tr>
<td>Create New Code Template</td>
<td>Copies a template file, $HPEESOF_DIR/modelbuilder/lib/cui_circuit.template to the current project networks directory with the name &lt;component&gt;.c. A warning message is displayed if a .c file exists with the same name in the current project's networks directory. If you continue, the .c file will be overwritten.</td>
</tr>
<tr>
<td>Open File</td>
<td>Brings up the Open file dialog box, so you can browse through your directory structure to select the pathname for the file you want to open. See “Open File Dialog” on page 1-9.</td>
</tr>
<tr>
<td>Save</td>
<td>Saves all current edits to the active file.</td>
</tr>
<tr>
<td>Save As</td>
<td>Brings up the Save As dialog box so you can save all current edits to a new file with a different name. See “Save As Dialog” on page 1-10.</td>
</tr>
<tr>
<td>Compilation Status</td>
<td>Displays the error/warning messages of the last compile and link process.</td>
</tr>
</tbody>
</table>
Building User-Compiled Analog Models

Table 1-2. User-Compiled Circuit Model Code Generation Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile Simulator/Compile Model</td>
<td>Generates the makefile (modelbuilder.mak), auto-generates the simulator boot procedure using the names of the object files, and executes the make process. If the selection “Recompile all out-of-object files” is checked in the Compile Options dialog box (see following description), any other .o(bj) files specified in the environment variable USER_MODEL_LINK_LIST will be recompiled if necessary, and linked into the executable. (This environment variable is located in the file de_sim.cfg in the current project directory.) Under the path $HPEESOF_DIR/modelbuilder/lib/hpeesof, the files hpeesofdebug.mak (or hpeesofopt.mak) and user.mak are copied to the current project's networks directory if they don't exist. (Either hpeesofdebug.mak or hpeesofopt.mak is copied, depending on the selected setting of “set debug flag on” in the Compile Options dialog box.) Do not change the makefile (hpeesofdebug.mak or hpeesofopt.mak). But you can edit the user.mak file. This button's label and corresponding function (compiling the simulator or only compiling the current model) are also determined by selections in the Compile Options dialog box.</td>
</tr>
<tr>
<td>Compile Options</td>
<td>Brings up the Compile Options dialog box, which allows you to set specifications for the compilation process. See “Compile Options Dialog” on page 1-11. If OK is selected, the files to be linked are saved in the environment variable USER_MODEL_LINK_LIST in the file de_sim.cfg in the current project directory.</td>
</tr>
<tr>
<td>OK</td>
<td>Choose this button when you have made all desired specifications in the tabs of the User-Compiled Circuit Model dialog box. The component information will be written to a file called &lt;component_name&gt;.ael and saved to the database. The program checks for a &lt;component_name&gt;.c file and will prompt you to generate one if such a file is not found. The dialog box is dismissed.</td>
</tr>
</tbody>
</table>
Open File Dialog

Enables you to browse through your directory structure to select the pathname for the code file that you want to open.

Table 1-3. Open File Dialog Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter box</td>
<td>Displays the currently selected directory.</td>
</tr>
<tr>
<td>Directories</td>
<td>List of accessible directories.</td>
</tr>
<tr>
<td>Files</td>
<td>List of user-defined code files in the currently selected directory</td>
</tr>
<tr>
<td>Selection</td>
<td>Type in a pathname for the directory where your user-defined code is stored or change directory paths in the Directories box until the correct path is shown here.</td>
</tr>
<tr>
<td>OK</td>
<td>Opens the selected code file and dismisses the dialog box.</td>
</tr>
<tr>
<td>Filter button</td>
<td>Change to the selected directory and display all user-defined code files from this directory in the Files box.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Cancel the selected pathname and dismiss the dialog box.</td>
</tr>
</tbody>
</table>

Table 1-2. User-Compiled Circuit Model Code Generation Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply</td>
<td>Choose this button when you have entered all specifications for a particular field in the Model Code tab. The specified data will be saved to the appropriate files and the dialog box will remain open.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Choose this button to cancel any data you have specified in the User-Compiled Circuit Model box. This will cancel all changes, dismiss the User-Compiled Circuit Model dialog box, and close the Schematic window (if it is open).</td>
</tr>
</tbody>
</table>
Save As Dialog

Enables you to browse through your directory structure to select the pathname for the code file that you want to open.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter box</td>
<td>Displays the currently selected directory.</td>
</tr>
<tr>
<td>Directories</td>
<td>List of accessible directories.</td>
</tr>
<tr>
<td>Files</td>
<td>List of user-defined code files in the currently selected directory.</td>
</tr>
<tr>
<td>Selection</td>
<td>Type in a pathname for the directory where you want to save the user-defined code file followed by the desired filename or change directory paths in the Directories box until the correct path is shown here, then type in the filename at the end of the path.</td>
</tr>
<tr>
<td>OK</td>
<td>Creates the selected code file as specified and dismiss the dialog box.</td>
</tr>
<tr>
<td>Filter button</td>
<td>Change to the selected directory.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Cancel the selected pathname and dismiss the dialog box.</td>
</tr>
</tbody>
</table>

Code Options (Analog/RF Models) Dialog

Enables you to specify certain specifications for the code, such as type of analysis function. Check the boxes for the items that you want to include. By default, no functions are checked. These values are not saved in the <component_h>.c file until you choose the Create New Code Template button.
**Compile Options Dialog**

Enables you to specify the option files in a particular project that are to be linked to or unlinked from the simulator executable.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Projects</td>
<td>Use the drop-down list to select the appropriate project from the list of available projects.</td>
</tr>
<tr>
<td>Objects Files in Project</td>
<td>Select the object files that you want to link to the project, then click the right-arrow button.</td>
</tr>
<tr>
<td>Object Files to Be Linked</td>
<td>Includes the previously compiled model files for the project shown. Select the object files that you want to unlink from the project, then click the left-arrow button.</td>
</tr>
<tr>
<td>Set debug flag on</td>
<td>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.</td>
</tr>
<tr>
<td>Create static executable</td>
<td>When you select this option, the compiler will link the selected models into an executable. The executable is created in the current project directory. You can also choose to place the new executable in a different location to use alternate executables. You can then add any of the symbols from the linked models to a schematic. Default is selected.</td>
</tr>
<tr>
<td>Create dynamic link model</td>
<td>When you select this option, enter a library name in the field labeled Dynamic link library name. The compiler will link the selected object files and create a model that can be distributed as a dynamically linked library (.dll) or shared library (.sl or .so). Default is unselected.</td>
</tr>
<tr>
<td>OK</td>
<td>Click to dismiss this dialog box. All object files in the box on the right of this box will be linked.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Cancels all specified data in this box and dismiss the dialog box only.</td>
</tr>
</tbody>
</table>
Specifying the Model Options

The Options tab enables you to specify a variety of options associated with the user-defined model.

Table 1-6. User-Compiled Circuit Model Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Description</td>
<td>Type in a descriptive name for the component. The default, as shown in the box, is User-Compiled Model. This will be the identifier used for this component when it is edited in the Edit Component dialog box, which is used for modifying component parameters in the Schematic window.</td>
</tr>
<tr>
<td>Component Instance Name</td>
<td>(Optional) The name specified in this box is used as a prefix in building a unique name (ID) for every user-defined item. It is part of the annotation displayed with the symbol representing the parameter subnetwork when placed in a design. The default is the name of the component.</td>
</tr>
<tr>
<td>Library Name.</td>
<td>(Optional) The name of the library where the component is located. The default is My Circuit Library. Type in a new name if desired.</td>
</tr>
<tr>
<td>Layout Artwork (Type)</td>
<td>(Optional) You can specify an artwork representation to be associated with the component in the Layout window. The available types are Synchronized, Fixed, AEL Macro, and None. None is the default.</td>
</tr>
<tr>
<td>Layout Artwork (Name)</td>
<td>(Optional) You can select the name of an artwork representation to be associated with the component in the Layout window if you select Fixed or AEL Macro in the Layout Artwork (Type) field.</td>
</tr>
<tr>
<td>Include in Bill of Materials</td>
<td>(Optional) if this box is checked, the instances of the user-defined component will be logged in the Bill of Materials. Default is not checked.</td>
</tr>
<tr>
<td>Use External Text Editor</td>
<td>If this box is checked, a text editor will start. The default editor used is the hpeesofedit editor. If you wish to access a different text editor, type in its pathname, then select the check-box. (The vi editor is not available.)</td>
</tr>
<tr>
<td>OK</td>
<td>Choose this button when you have made all desired specifications in the tabs of the User-Compiled Circuit Model dialog box. The component information will be written to a file called &lt;component_name&gt;.ael and saved to the database. The program checks for a &lt;component&gt;.c file and will prompt you to generate one if such a file is not found. The dialog box is dismissed.</td>
</tr>
</tbody>
</table>
Opening an Existing Model

To work on an existing user-compiled model, open the project where the model exists. Use the menu command Tools > User-Compiled Model > Open User-Compiled Model. Select the model from the browser.

Table 1-7. Open User-Compiled Model Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Projects</td>
<td>Use the drop-down list to select the project where the user-compiled model that you want to open is stored.</td>
</tr>
<tr>
<td>Models in Project</td>
<td>Click the name of the model that you want to open.</td>
</tr>
<tr>
<td>OK</td>
<td>Click this button to open the selected model and dismiss this dialog box.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Click this button to cancel the selected model and dismiss the dialog box.</td>
</tr>
</tbody>
</table>

Table 1-6. User-Compiled Circuit Model Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply</td>
<td>Choose this button when you have entered all specifications in the Option tab. The specified data will be saved to the appropriate files and the dialog box will remain open.</td>
</tr>
<tr>
<td>Cancel</td>
<td>Choose this button to cancel any data you have specified in the User-Compiled Circuit Model dialog box. This will cancel all changes, dismiss the User-Compiled Circuit Model dialog box, and close the Schematic window (if it is open).</td>
</tr>
</tbody>
</table>
Deleting a Model

To permanently delete an existing user-compiled model, open the project where the model exists. In the Schematic window, choose the menu command Tools > User-Compiled Model > Delete User-Compiled Model.

In the dialog’s Active Projects field, use the drop-down list to select the appropriate project from the list of available projects. From the Models in Project box, click the name of the model(s) that you want to delete. Then choose one of the following commands:

- Click Cancel to dismiss the dialog box only.
- Click Apply to delete the selected model(s) and leave the dialog box open.
- Click OK to delete the selected model(s) and dismiss the dialog box.

The program prompts you for confirmation before deleting.

Linking Model Files to the Simulator

To link or unlink selected object files to the simulator, move the model files into the project’s networks directory, then open the project where the model files exist. In the Schematic window, choose the menu command Tools > User-Compiled Model > Link User-Compiled Model. Previously compiled user models can be linked without recompiling.

**Note**  Be sure to verify the link type which is set when specifying the model’s compile options. The compile options are available from the Model Code tab on the User-Compiled Circuit Model dialog (Schematic window: Tools > User-Compiled Model > Open User-Compiled Model). See “Compile Options Dialog” on page 1-11.

The Link Status box displays error/warning messages from the previous link. To dismiss the dialog box only, select Cancel. To add or delete options to be linked, choose the Link Options button. This opens the Link Options dialog box (see “Link Options” on page 1-15). After you verify the link type specifications, and set the link options, choose the Link Simulator button to link all selected object files into a new executable. The new executable will be placed in the projects directory (if an existing executable was already there, it will be renamed).
Link Options

The Link Options dialog enables you to specify the object files from a particular project to be linked or unlinked. From the Active Projects field, select an active project from the drop-down list. The box on the left, Object Files in Project, shows object files that exist in the selected project. The box on the right, Object Files to be Linked, specifies object files to be linked. To add to the list of files to be linked, select the appropriate file to be linked from the left box, then click the right-arrow button. To remove files from the list to be linked, select the appropriate file from the right box, then click the left-arrow button. If you wish to cancel changes that you have entered and dismiss the box, click Cancel. If you wish to activate the changes you have made in the box, click OK.

Message: Confirm Files Out-of-Synch

This message occurs when you attempt to dismiss the active dialog box without having linked recently created files that are more current than the existing executable. If you wish to dismiss the box without linking the new models, click Yes. Otherwise, click No.
Example - Creating a New User-Compiled Analog Model

This section uses an example to demonstrate the steps that are required to create a new analog user-compiled model using the Advanced Design System's Model Development Kit. The following procedures are shown:

- Starting a new model
- Defining the model parameters
- Creating the model symbol and adding pins
- Writing and compiling the model
- Setting options

Starting a User-Compiled Model

1. Verify that Analog/RF is the current design type.
2. From the Schematic window, choose Tools > User-Compiled Model > New User-Compiled Model.
3. Enter a name (in this example, MyModel). Click OK.
4. In the dialog box that appears, select the model type:
   - Linear can have linear elements only
   - Nonlinear can have both linear and nonlinear elements

Because this example model is a nonlinear diode (with two external pins and one internal node), select Nonlinear for the circuit type.

5. Click OK and the User-Compiled Circuit Model dialog box appears.

**Defining the Model Parameters**

Use the Parameters tab to define input parameters. Each parameter must be specified by name in the Parameter Name field and a value type (integer, string, real, etc.) assigned to it.

*Hint*  It is usually a good idea (but not required) to specify a default value in the Default Value field.

Optionally you can specify a Parameter Type (unit) and description, as well as a variety of parameter attributes. The Parameter Type controls the scale factor label that is displayed in the dialog box during subsequent editing in the Edit Component Parameters dialog box. The text you enter in the Parameter Description field is displayed in the component parameter dialog box that appears when you edit a component/instance).
Building User-Compiled Analog Models

The remaining optional parameter attributes can be used as follows:

- **Display parameter on schematic**—defines whether or not the parameter appears on the schematic
- **Optimizable**—controls whether or not to allow this parameter to be optimized
- **Allow statistical distribution**—controls whether or not this parameter is allowed a statistical distribution when used in conjunction with the simulator's statistical features

**Note** Once the model is compiled and inserted in the design, these attributes cannot be modified.

If your parameters are similar to those in a supplied component, you can copy and then modify them.

To copy the parameters of a supplied component:

1. Click **Copy Parameters From**.
2. When the library browser appears, select the appropriate library and component, in this example, Devices-Diodes > Diode_Model (PN-Junction Diode Model).
3. The parameter list is updated and can now be modified.

**Note** Copying parameters from a component will also copy the default symbol, which can then be modified as desired. The symbol can be seen when you click Create/Edit Symbol and Pins from the Model Code tab.
Creating the Model Symbol

Once the parameters are defined, the next step is to generate or create the schematic symbol (or modify the existing one if you used the Copy Parameters From option in the Parameters tab). This is done from the Model Code tab.

Click Create/ Edit Symbol and Pins to switch to symbol view in the Schematic window. If you copied parameters from a supplied component (as in this example), the associated symbol will be there, otherwise a default symbol is generated, based on the number of ports on your design. Add any text or graphics as well as the proper number of external pins. The program will determine how many external nodes exist by the number of pins on the symbol page. You will be able to access the pins in your code via auto-generated macros. You can also modify the symbol graphics and number of pins at any time in the model generation process. Figure 1-1 shows the symbol after modification. The bounding box was deleted, external pins added, and text inserted.

![MyModel]

Figure 1-1. Symbol after modification

When done defining the symbol, save the file and (optionally) close the window.
Setting Options

Additional options can be set on the Options tab:

- A description for the model
- The library the model should be part of
- Any associated layout artwork and macros
- An attribute to have the component counted in the Bill of Materials
- Specifying an alternate text editor

By setting an alternate text editor you can use something other than the limited text editor accessed from the dialog box. Note that if you choose to use an external editor, it is up to you to save your files before compiling.
Building User-Compiled Analog Models

Creating the Code and Compiling the Model

The code generation is done from the Model Code tab.

To set the desired options prior to generating:

Click Code Options. In the dialog box that appears, you can select those functions available, as well as set the value for the number of internal, nonlinear nodes. The Transient Function provides a field for No. of internal nodes for the transient topology. The field appears for both linear and nonlinear models. The No. of nonlinear internal nodes field displays only for nonlinear models.

![Code Options dialog box](image)

**Note** The program will generate a file (named MyModel_h.c in this example) which has macros defined to either enable or disable the functions. This file is auto-generated for each compile, so you can modify these options if you decide to add another function. The user-edited file is not referenced in the Makefile, it is '#nduded' from the MyModel_h.c file.
A template is provided to assist you in quickly creating your code. It is self-documenting and provides prototypes of all of the functions. (Note that functions are prototyped so that they at least return a value, but this does not guarantee that they will not cause simulation problems.)

To copy the template to your working model file:

1. Click Create New Code Template. In this example, the file will be called MyModel.c.
2. When the template code appears, edit as necessary to add the correct functionality for your model.
3. If using an external text editor, save the file.

**Hint** To edit an existing file, click Open File.

To set the compiling options prior to compiling:

Click Compile Options. Note that the current file is already in the Object Files to be Linked list box on the right-hand side. You can optionally select other models to be linked into the simulator. At this time you can also set the following options as desired:

- **Set debug flag on**
  
  When this option is selected, the appropriate flag is passed to the compiler to include the debug information in the object code. You will then be able to step through your code when the debugger is invoked. Refer to your debugger documentation for details.

- **Create static executable**
  
  When this option is selected, the compiler will link the selected models into an executable. The executable is created in the current project directory. If you want to place the new executable in a different place, please see “Using an Alternative hpeesofsim Executable” on page 1-56. You can then add any of the symbols from the linked models to a schematic.
Building User-Compiled Analog Models

• Create dynamic link model

When this option is selected, enter a library name in the field labeled Dynamic link library name. The compiler will create a model that can be distributed as a dynamically linked library (.dll) or shared library (.sl or .so). For information about using a dynamic link model, see “Accessing Dynamically Loaded Devices” on page 1-54.
To compile the code:

Click **Compile**. The status is displayed in the Compilation Status pane. If the compile and link process was successful, a new simulator executable will have been generated in the current project directory. (If a prior executable existed, it will have been renamed.)
Building User-Compiled Analog Models

Once compiled, the model is immediately available in the design environment. From a Schematic window open the Library browser. The model can be found available in the My Circuit Library (unless you supplied a unique library name in the Options tab).
Select the component and place it on your schematic, just like any other built-in component.
Building User-Compiled Analog Models

Characteristics of User-Compiled Elements

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 any 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.
Creating Circuit Elements Interface Definitions and Declarations

Interfacing to the simulator code requires the use of certain ADS defined public C symbols in user-defined element modules. The remainder of this chapter 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.

```c
#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.

```c
/* * define some physical constants */
#define BOLTZ 1.380658e-23
#define CHARGE 1.60217733e-19
#define CTOK 273.15
#define NOISE_REF_TEMP 290.0 /* 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.

```c
#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.

```c
#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 */
```
Linear response modeled in the frequency domain is complex, so the \texttt{COMPLEX} type is used for admittance (\(Y\)), scattering (\(S\)), and noise current-correlation parameters.

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

Each element parameter has a specific type.

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

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

```c
typedef struct {
  char *keyword;
  DataTypeE dataType
} UserParamType;
```

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

```c
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 */
  }value;
} UserParamData;
```
Building User-Compiled Analog Models

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

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

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.

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.

The `compute_y` function must load the nodal admittance matrix parameters at frequency omega radians/sec into the passed `ypar` array. When frequency is zero, a value of 1 is passed to omega. This function can call `ee_compute_y` (described later) to use another element's admittance parameters.

The `compute_n` function must load the normalized nodal noise current correlation parameters (Siemens, normalized to FOUR_K_TO) into the passed `nCor` array at frequency omega 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.

The `post_analysis` function, called before processing a new circuit, is generally used for cleanup operations such as freeing memory or deleting temporary files. This function is called for all types of analysis. This function is called once for each occurrence of an element, so it must be written to properly manage this situation.

A nonlinear element must contain additional device information in a static area of type `UserNonLinDef` (described later); the pointer `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.

---

1-32 Characteristics of User-Compiled Elements
A transient response for an element can be defined in a structure of type UserTranDef (described later); the pointer tranDef must point to the structure. A transient response function can be defined for either a linear or nonlinear element.

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.
    * NULL for nonlinear element */
    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);

    /* post-analysis: called before new netlist is parsed */
    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 */
};
A nonlinear element response is characterized by the functions in the UserNonLinDef type.

numIntNodes is the number of nodes internal to the element. In its model, the element code must compute the contributions at all of its pins, which are ordered by external followed by internal pins. If a UserTranDef type is defined for the element, the numIntNodes in that structure must match this definition.

analyze_lin must load 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.

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

\[
\text{r}_n(t) = f(v_0(t), v_1(t), \ldots, v_{P-1}(t), v_k(t-*k), v_l(t-*l), \ldots) \quad \text{where } r_n \text{ is the pin current or charge response,} \\
*_{k, l} \ldots \text{are ideal delays, independent of the voltages. These responses and their} \\
\text{derivatives with respect to voltage (nonlinear conductances, capacitances) must be} \\
\text{computed and loaded using the } \text{add\_nl\_iq} \text{ and } \text{add\_nl\_gc} \text{ functions, respectively. Note} \\
\text{that the derivatives help the simulator to converge to a solution, but do not affect the} \\
\text{steady-state nonlinear response—therefore they may work even if not exact.} \\
\text{However, under certain simulation conditions in-exact derivatives may cause} \\
\text{convergence problems. However, for noise analysis they should be accurate, and for} \\
\text{convergence they should be continuous.} \\
\]

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.

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.

```c
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);
};
```

A transient response function can be defined for any element (linear or nonlinear) by defining the functions in the `UserTranDef` type shown.

The `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:

\[ r_n(t) = f(v_0(t), v_1(t), ..., v_{P-1}(t)) \]

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);
};
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.

```c
typedef struct _UserInstDef UserInstDef;
struct _UserInstDef {
    char *tag; /* item name */
    UserElemDef *userDef; /* access to user-element definition */
    UserParamData *pData; /* item's parameters */
    void *eeElemInst; /* EEsof's element item */
    void *eeDevInst; /* EEsof's nonlinear device item */
    void *seniorData; /* data allocated/managed/used only by Senior module (arbitrary) */
};
```

The `get_params` function, below, loads the passed item `eeElemInst` parameter values into `pData`, which must be big enough to store all parameters. It is used to obtain referenced item (such as model substrate) parameters. Note that user-defined item parameters are already available in the `UserInstDef.pData` array, so there is no need to call `get_params` for user item parameters. It returns `TRUE` if successful, `FALSE` otherwise.

```c
extern boolean get_params (IN void *eeElemInst, OUT UserParamData *pData);
```

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.

```c
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.

```c
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.
Building User-Compiled Analog Models

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.

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

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

```c
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.

```c
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 omega.

```c
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.

```c
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.

```c
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);
Building User-Compiled Analog Models

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 \( \text{ind1} \) and the inductor \( \text{ind2} \).

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

\( \text{add_tr_inductor()} \) or \( \text{add_tr_lossy_inductor()} \) must be added before the mutual inductance is added. \( \text{ind1} \) and \( \text{ind2} \) are the values returned from \( \text{add_tr_inductor()} \) or \( \text{add_tr_lossy_inductor()} \). Here is an 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;
```
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 $z_0$ Ohms and the propagation delay time of the line is $t_d$ 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.

```c
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);
```

The function below must be called by user-defined element code (possibly from a pre_analysis function) that wants to use an element built in ADS. 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.

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

These functions allow access to Advanced Design System elements for linear and noise 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).

```c
extern boolean ee_compute_y (INOUT void *eeElemInst, IN UserParamData *pData, IN double omega, OUT COMPLEX *yPar);
extern boolean ee_compute_n (INOUT void *eeElemInst, IN UserParamData *pData, IN double omega, IN COMPLEX *yPar, OUT COMPLEX *nCor);
```

The function below must be called by user-defined element code (possibly from a post_analysis function) for every ee_pre_analysis call to free memory allocated for the Advanced Design System item eeElemInst.

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

This returns a pointer to the UserInstDef user-defined item if eeElemInst is indeed an item of a user-defined element, NULL otherwise.

```c
extern UserInstDef *get_user_inst (IN void *eeElemInst);
```

The function converts between S- and Y-parameters. If direction is 0, it computes S-parameters into outPar using inPar as Y-parameters. If direction is 1, it computes Y-parameters into outPar using inPar as S-parameters. rNorm is the S-parameter.
Building User-Compiled Analog Models

normalizing impedance in ohms, and size is the matrix size. It returns TRUE if successful, FALSE otherwise.

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

Series IV Functions

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_voltunit(eeElemInst) 1.0 /* voltage unit */
#define get_watt(eeElemInst, power) (power) /* power unit */
Referencing Data Items

A user-defined element parameter can be a reference to an ADS or a User-Defined item. Use the get_params function to obtain the referenced item's parameters.

As an example, if you are creating a microstrip element and need an MSUB reference, the third parameter, for example, can be:

```
{ "MSUB", MTRL_data }
```

Then 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:

- Er Relative dielectric constant.
- Mr Relative permeability.
- H (m) Substrate thickness.
- Hu (m) Cover height.
- T (m) r Conductor thickness.
- Cond (Siemens/m) Conductor conductivity.
- TanD Dielectric Loss Tangent.
- Rough Conductor surface roughness.

into `mData[0...7]` locations. The `mData` array must be dimensioned large enough to hold all the referenced item's parameters. If a parameter value is not set or available, the `dataType` enum value will be NO_data.

If the referenced item (using the third parameter again) is an item of a user-defined Data Item, then you can obtain a pointer to the user item as follows:

```
refInst = get_user_inst(userInst->pData[2].value.eeElemInst)
```

The function `get_user_inst` will return NULL if the passed argument is not a user-defined item.
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");
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:

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

To allocate a pseudo item and do any pre-analysis processing such as data file reading, the user-defined element must first call this function. The second argument, `UserParamData *pData`, must contain the data of correct type and in the order expected by the ADS element. The values must be in SI units where applicable. 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`.

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

The function below obtains N×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.

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

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

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

This function must be called for each pseudo item created by each `ee_pre_analysis` call by the user-defined element. This frees up the memory allocated for the ADS item. It is usually called from the user-defined element's `post_analysis` function. It returns TRUE if successful, FALSE otherwise.
Building User-Compiled Analog Models

**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`:

```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` from `boot_abc`:

```c
extern boolean load_elements (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.
Porting Libra Senior to the ADS 1.0 Model Builder Interface

The ADS 1.0 Model Builder interface is designed to allow Series IV-Libra Senior models to be easily ported. Most models will require recompiling using the supplied makefiles. However, several features will require code modification if they were used:

- Data Items
- Default Units
- Substrates
- Calls to built-in elements

**Data Items**

The Data Item is replaced by references to an AEL expression, or equation. Users can still access default values, however, the ‘*‘ value is no longer supported and must be replaced by references to built-in values. For example, the Series IV Data Items can be set via comparable VAR expressions:

```plaintext
_DEFAULT_TEMP = 290.15
_DEFAULT_REF = 50
```

Instead of using the ‘*‘ value to indicate the default, the parameter should be left blank. If the value on the schematic is set to blank the simulator defaults to the Series IV default values:

<table>
<thead>
<tr>
<th></th>
<th>Temperature</th>
<th></th>
<th>Resistance</th>
<th></th>
<th>Loss tangent</th>
<th>Permeability &amp; Mag loss tangent</th>
<th>Conductivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEMP</td>
<td>TEMPERATURE</td>
<td>TEMP</td>
<td>R</td>
<td>R</td>
<td>TAND</td>
<td>PERM</td>
<td>SIGMA</td>
</tr>
<tr>
<td>RREF</td>
<td>Reference</td>
<td>REF</td>
<td>Reference</td>
<td>R</td>
<td>TANM</td>
<td>PERM</td>
<td>SIGMA</td>
</tr>
<tr>
<td>TAND</td>
<td>Dielectric</td>
<td>TAN</td>
<td>Dielectric</td>
<td>R</td>
<td>MUR, TANM</td>
<td>PERM</td>
<td>SIGMA</td>
</tr>
<tr>
<td>PERM</td>
<td>Permeability</td>
<td>PERM</td>
<td>Permeability</td>
<td>MUR</td>
<td>TANM</td>
<td>PERM</td>
<td>SIGMA</td>
</tr>
<tr>
<td>SIGMA</td>
<td>Dielectric</td>
<td>SIGM</td>
<td>Conductivity</td>
<td>SIGM</td>
<td>TANM</td>
<td>PERM</td>
<td>SIGMA</td>
</tr>
</tbody>
</table>
Building User-Compiled Analog Models

**Default Units**

The concept of Default Units is not used in ADS. Instead, each parameter on a component can have a scale factor. The simulator's parsing utility replaces the parameter value+scale factor with the proper number (e.g., 1.23GHz is supplied to the user's program as 1.23e9). Any calls to `get_unit` functions are still supported; they merely return a value of 1.

Note that the scale label is ignored:

```
Resistance = 100 kOhm
  = 100 * 1e3
  = 100 kBricks
  = 100 kHz
```

All calls to any of the units functions are all equivalent (the user's code will receive a value of 1e5).

**Substrates/Built-In Models**

The substrate use model has changed considerably in ADS and requires users to modify their calls to any substrate components. Certain parameters have been moved from Data Items to the instance, others to the substrate component. The order and name of the parameters has also changed. Conductivity is specified instead of resistivity.

User calls to built-in models may also have changed, depending on the element. Users will have to reconfirm parameter ordering.

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`).
AEL Changes

Series IV AEL may require substantial modification. For example, AEL references to Series IV forms (rvopt, etc.) should be changed to 'StdFormSet.' It may be expedient to use the Model Development Kit to generate the AEL file rather than edit a Series IV file.

```c
create_item("U2PA","User-Compiled Model","U2PA",16,-1,NULL,"Component Parameters","",&d:t%44?0%:31%C:_net%c%;%e%b%8%29%:%30%p:%k%&[li]%;&p%;%e%e","U2PA","%t%b%38%:%n%39?all_parm%A:%30%s:%k%&[li]%;&s%;%e%e;","U2PA",3,NULL,64,
create_parm("R1","Pin 1 to GND Resistance ",68608,"StdFormSet",-1,
prm("StdForm","50")),
create_parm("R2","Pin 1 to Pin 2 Resistance",68608,"StdFormSet",-1,
prm("StdForm","50")),
create_parm("R3","Pin 2 to GND Resistance",68608,"StdFormSet",-1,
prm("StdForm","50"))
```
Building User-Compiled Analog Models

Managing Model Files

The Analog Model Development Kit interface uses several files as a means of managing the C-code and the compile and link process. The automated compile actions described here must be completed before building user-compiled models from the command line.

The program copies certain files from the $HPEESOF_DIR/modelbuilder/lib directory and generates additional files depending on selected compiler and link options. The files are placed in the ./networks directory under the current local project directory. As an example, for a model file called MyModel, the following file management takes place when the Compile button is clicked:

- If Create New Code Template is selected, `cui_circuit.template` is copied to `MyModel.c`.
- One of the following `hpeesof*.mak` files is always copied. The specific file depends on the compiler options settings:
  - `hpeesofdebug.mak` if the debug flag is on, and the link type is static executable.
  - `hpeesofopt.mak` if the debug flag is off, and the link type is static executable.
  - `hpeesofdynamic.mak` if the link type is dynamic link model.
- `userindx.c` and `user.mak` are copied if they are not in the project directory.
- `MyModel_h.c`, `cui_indx.c`, and `modelbuilder.mak` are created if they do not already exist. Existing files are overwritten if they are out-of-date relative to dialog box settings.
- If Create dynamic link model is the link type, `MyModel_DYNAMIC.cxx` is created.

The Makefile compiles the `MyModel_h.c` file which is autogenerated when the Compile button is clicked and the file is out-of-date relative to dialog box settings. It includes the header files and the user's code.

During the compile process the Makefile sources the `hpeesof*.mak` file. This file includes the autogenerated `modelbuilder.mak` file and the user-editable `user.mak` file. The Model Development Kit interface executes the make process. The following figure shows the C-code file relationship:
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/bin must be in $PATH.

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 ./networks directory under the current local project directory. You may also need to modify some of the files first. Table 1-8 lists the required files and whether modification is needed.

---

Figure 1-2. C-code File Relationship

MyModel.h.c

```c
#include "userdefs.h"

The program generates macros to connect the AEL with the C-code as well as to simplify c addresses.

#include "MyModel.c"
boot_senior_MyModel();
```

Userdefs.h

All of the necessary prototypes are defined here.

MyModel.c

A template file is used to help the user write the code. From then on, only the user modifies this file.

This file is the model.
Creating a Dynamic Link Model

To create a dynamic link model, use the file `hpeesofdynamic.mak`. In addition, 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 for your needs. 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 `.networks` directory in the current local project directory. The template must be customized since it cannot be used as it is. After the required files are in place, run the compile command.

The compile command is:

```
hpeesofmake -f hpeesofdynamic.mak [debug=1]  
```

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

<table>
<thead>
<tr>
<th>File Name</th>
<th>Required Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpeesofdebug.mak</td>
<td>No modification is needed.</td>
</tr>
<tr>
<td>hpeesofopt.mak</td>
<td>No modification is needed.</td>
</tr>
<tr>
<td>hpeesofdynamic.mak</td>
<td>No modification is needed.</td>
</tr>
<tr>
<td>modelbuilder.mak</td>
<td>Customization is usually needed.</td>
</tr>
<tr>
<td>user.mak</td>
<td>No modification is needed except to customize the compile or link options.</td>
</tr>
<tr>
<td>cui_idx.c</td>
<td>No modification is needed.</td>
</tr>
<tr>
<td>userindx.c</td>
<td>No modification is needed.</td>
</tr>
<tr>
<td>userdefs.h</td>
<td>No modification is needed.</td>
</tr>
<tr>
<td>MyModel_DYNAMIC.cxx</td>
<td>Required only for dynamic link user-compiled model. This file is only an example; modification is required before using.</td>
</tr>
</tbody>
</table>
Creating a Static Executable

To create a static executable, use hpeesofdebug.mak (to compile your source code with debug information) or hpeesofopt.mak (no debug information). After the required files are in place, run the compile command.

The command to compile with debug information is:

```
hpeesofmake -f hpeesofdebug.mak <target>
```

where `<target>` is one of the following:

- `link_only` Link without compiling anything.
- `compile_and_link` Compile the active file and link everything.
Building User-Compiled Analog Models

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 project in which it is created. The procedure that follows describes how to make a device available to other projects or users.

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

   .. network
   $HOME/hpeesof/circuit/lib.$ARCH
   $HPEESOF_DIR/custom/circuit/lib.$ARCH
   $HPEESOF_DIR/circuit/lib.$ARCH

   **Note** To determine $ARCH, run $HPEESOF_DIR/bin/hpeesofarch. For ADS 2002, the program returns the ARCH values win32, hpux10, aix4, and sun57. They may change in the future.

   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, see the chapter “ADS Simulator Input Syntax” in the Circuit Simulation documentation.
Building User-Compiled Analog Models

Using an Alternative hpeesofsim Executable

If you have compiled a user-compiled model and created a new hpeesofsim executable in your project directory, you may want to access the new executable without replacing the built-in executable installed with ADS. You can place the new executable in a different location, then configure ADS to use it instead.

The file $HPEESOF_DIR/config/hpeesofsess.cfg contains the configuration variable $HPEESOFSIM_BIN. Set this variable to point to any directory where you put your new hpeesofsim executable. The default configuration is:

HPEESOFSIM_BIN=hpeesofsim

To set your own configuration:

1. Create a new text file in the location required for site-wide or user configuration:
   
   $HPEESOF_DIR/custom/config/hpeesofsess.cfg  (site-wide)

   or

   $HOME/hpeesof/config/hpeesofsess.cfg  (user)

2. In the configuration file, set $HPEESOFSIM_BIN to your executable’s location. An example statement in the hpeesofsess.cfg file might be:

   HPEESOFSIM_BIN=c:/temp/hpeesofsim.exe

3. Save the configuration file and restart ADS.

---

Note: Use forward slashes in the configuration file, even on the PC.

The executable must be called hpeesofsim on UNIX, and hpeesofsim.exe on the PC.

---

1-56   Using an Alternative hpeesofsim Executable
Chapter 2: Creating Linear Circuit Elements

This chapter 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 Figure 2-1. Although this example shows 2-port S-parameters, the technique is the same for elements with a greater number of ports.

![Figure 2-1. 2-port network](image)

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

\[
S_{11} = \frac{Z_1 - Z_0}{Z_1 + Z_0} \\
S_{22} = \frac{Z_2 - Z_0}{Z_2 + Z_0}
\]

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 $Z_0$
Creating Linear Circuit Elements

For example, consider a grounded pi-section resistive attenuator as shown in Figure 2-2. Inserting Figure 2-2 into the two-port network in Figure 2-1 results in Figure 2-3.

Figure 2-2. Schematic for pi-section resistive attenuator

---

2-2 Deriving S-Parameter Equations
Using Figure 2-3, the following relations are defined:

$$YA1 = 1.0/R3 + 1.0/ZO$$

$$ZA1 = 1.0/YA1$$

$$ZB1 = R2 + ZA1$$

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

$$YA2 = 1.0/R1 + 1.0/ZO$$

$$ZA2 = 1.0/YA2$$

$$ZB2 = R2 + ZA2$$

From the definition of $Z1$ and $Z2$:

$$Z1 = (R1 \cdot ZB1)/(R1 + ZB1)$$

$$Z2 = (R3 \cdot ZB2)/(R3 + ZB2)$$

The $S$-parameters are obtained from the following equations:

$$S_{11} = (Z1 - ZO)/(Z1 + ZO)$$

$$S_{22} = (Z2 - ZO)/(Z2 + ZO)$$

$$S_{12} = S_{21} = (2.0 / ZO)/(1.0 / ZA1 + 1.0 / ZA2 + R2 / (ZA1 \cdot ZA2))$$

These basic equations are sufficient to write the C function for the element.
Creating Linear Circuit Elements

Deriving Y-Parameter Equations

Y-parameters equations can be used to describe a user-defined element as an alternative to S-parameter equations. Figure 2-4 shows Y-parameters for a resistor connected between two ports; Y-parameter definitions follow the figure.

![Figure 2-4. Y-parameters for a 2-port resistor connection](image)

In general,

\[
Y_{11} = \frac{1}{V_{1}} \bigg|_{V_2 = 0} \quad \quad \quad Y_{21} = \frac{1}{V_{1}} \bigg|_{V_2 = 0} \\
Y_{12} = \frac{1}{V_{2}} \bigg|_{V_1 = 0} \quad \quad \quad Y_{22} = \frac{1}{V_{2}} \bigg|_{V_1 = 0}
\]

With \( V_2 \) equal to zero, \( V_1 = I_1R \), which is also equal to \(-I_2R\). \( Y_{11} \) reduces to \( 1/R \) and \( Y_{21} \) to \(-1/R\). Setting \( V_1 \) to zero, \( V_2 = I_2R = -I_1R \). 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:

```c
/***************************************************************************
/ # 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 |
/ *      | Y | = |-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
/***************************************************************************
/ */
```

Deriving Y-Parameter Equations 2-5
Creating Linear Circuit Elements

**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):

   ```c
   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:

   ```c
   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):

```c
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:

```c
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 $2 \times 2$ COMPLEX noise correlation matrix required by `compute_n` can be obtained using the provided function:

```c
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” on page 2-6 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.
Creating Linear Circuit Elements

The corresponding c-headers are automatically generated:

```c
#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 Figure 2-3 provide the basis for the needed variables. The equations are repeated here:

- \( YA_1 = \frac{1.0}{R_3} + \frac{1.0}{Z_0} \)
- \( ZA_1 = \frac{1.0}{YA_1} \)
- \( ZB_1 = R_2 + ZA_1 \)
- \( YA_2 = \frac{1.0}{R_1} + \frac{1.0}{Z_0} \)
- \( ZA_2 = \frac{1.0}{YA_2} \)
- \( ZB_2 = R_2 + ZA_2 \)
- \( Z_1 = \frac{R_1 \times ZB_1}{R_1 + ZB_1} \)
- \( Z_2 = \frac{R_3 \times ZB_2}{R_3 + ZB_2} \)
- \( S_{11} = \frac{Z_1 - Z_0}{Z_1 + Z_0} \)
- \( S_{22} = \frac{Z_2 - Z_0}{Z_2 + Z_0} \)
- \( S_{12} = S_{21} = \frac{2.0}{Z_0} / (1.0 / ZA_1 + 1.0 / ZA_2 + R_2 / (ZA_1 \cdot ZA_2)) \)

The resulting declarations are:

```c
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 $Y_{A1}$ and $Y_{A2}$ demonstrate the need to check data values. If $R_1$ or $R_3$ 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, $\text{EPS}$ 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}.

```c
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); /* S_{11} 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); /* S_{22} */
S[2].real = S[1].real = (2.0/ZO) /
(1.0/ZA1 + 1.0/ZA2 + R2/(ZA1 * ZA2));

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.

```c
#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×N Y-parameters of an N-terminal element and temperature to compute the N×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 Chapter 1, Building User-Compiled Analog Models.
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:

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

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

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

where:

\[A\,' = A = \cos\left(\frac{2\pi L}{\lambda}\right)\]
\[C\,' = CZ_o = j\left(\frac{Z}{Z_o}\right) \sin\left(\frac{2\pi L}{\lambda}\right)\]
\[B\,' = \frac{B}{Z_o} = j\left(\frac{Z}{Z_o}\right) \sin\left(\frac{2\pi L}{\lambda}\right)\]
\[D\,' = D = \cos\left(\frac{2\pi L}{\lambda}\right)\]

and

\[\Delta = A\,' + B\,' + C\,' + D\,'.\]

Define \(\beta\) as \(\frac{2\pi}{\lambda}\). Then \(\beta L = \frac{2\pi L}{\lambda}\).
Creating Linear Circuit Elements

When the mathematical equations are worked out, this leaves:

\[ S_{11} = S_{22} = \frac{j \sin(\beta L) \left( \frac{Z}{Z_0} - \frac{Z_0}{Z} \right)}{2 \cos(\beta L) + j \sin(\beta L) \left( \frac{Z}{Z_0} + \frac{Z_0}{Z} \right)} \]

and

\[ S_{12} = S_{21} = \frac{2}{2 \cos(\beta L) + j \sin(\beta L) \left( \frac{Z}{Z_0} + \frac{Z_0}{Z} \right)} \]

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:

\[ S_{11} = S_{22} = \frac{j \sin(\beta L) \left( \frac{Z}{Z_0} - \frac{Z_0}{Z} \right)}{2 \cos(\beta L) + j \sin(\beta L) \left( \frac{Z}{Z_0} + \frac{Z_0}{Z} \right)} \]

\[ \cdot \frac{2 \cos(\beta L) - j \sin(\beta L) \left( \frac{Z}{Z_0} + \frac{Z_0}{Z} \right)}{2 \cos(\beta L) - j \sin(\beta L) \left( \frac{Z}{Z_0} + \frac{Z_0}{Z} \right)} \]
This multiplication yields:

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

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

and

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

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

This multiplication yields:

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

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

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 $Z = 50 \, \Omega$, $Z_0 = 50 \, \Omega$, and $L = \lambda$. Then $\beta L = 2 \pi$. This yields:

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

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

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

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

Another example: $Z = Z_0 = 50 \, \Omega$, $\beta L = \pi/2$ (90°)

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

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

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

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

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
- \( \eta \) = characteristic impedance of dielectric = \( \frac{\sqrt{\mu_0}}{\sqrt{\varepsilon}} \)
- \( B \) = outside diameter
- \( A \) = inside diameter

The propagation constant is defined by:

\[ \beta = \frac{2\pi}{\lambda} \text{ where } \lambda = \frac{\text{speed of light in vacuum}}{\sqrt{E_r \cdot \text{FREQ}}} \]

Therefore, the required parameters are \( A \), \( B \), \( L \), and \( E_r \).
Creating Linear Circuit Elements

The U2PB example in U2PB.c is an implementation of the above. The relevant defining data structures are shown below:

```c
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 $a$, $b$, $\text{len}$, and $k$ values are obtained from the circuit through the automatically defined macros:

\begin{align*}
  a &= A_P; \\
  b &= B_P; \\
  \text{len} &= L_P; \\
  k &= K_P;
\end{align*}

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.

\begin{verbatim}
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;
}
\end{verbatim}
Creating Linear Circuit Elements

**Calculating Remaining Expressions**

The impedance and wave number are then calculated:

\[ \eta = \sqrt{\mu_0 / \varepsilon_0 / k} \]
\[ v_{\text{phase}} = 1.0 / \sqrt{\mu_0 \cdot \varepsilon_0 \cdot k} \]
\[ \beta_t = \omega \cdot \text{len} / v_{\text{phase}} \]
\[ z = \eta \cdot \log(b / a) / 2.0 / \pi \]

The remaining expressions calculate the S-matrix:

\[ z_{zo} = z / \text{Zo} \]
\[ z_{oz} = \text{Zo} / z \]
\[ \text{arg1} = z_{zo} - z_{oz} \]
\[ \text{arg2} = z_{zo} + z_{oz} \]
\[ \text{denom} = 4.0 \cdot \text{sqr}(\cos(\beta_t)) + \text{sqr}(\sin(\beta_t)) \cdot \text{sqr}(\text{arg2}) \]
\[ \text{res11} = \text{sqr}(\sin(\beta_t)) \cdot (\text{sqr}(z_{zo}) - \text{sqr}(z_{oz})) / \text{denom} \]
\[ \text{im}s11 = 2.0 \cdot \sin(\beta_t) \cdot \cos(\beta_t) \cdot \text{arg1} / \text{denom} \]
\[ \text{res21} = 4.0 \cdot \cos(\beta_t) / \text{denom} \]
\[ \text{im}s21 = -2.0 \cdot \sin(\beta_t) \cdot \text{arg2} / \text{denom} \]
\[ S[3].\text{real} = S[0].\text{real} = \text{res11}; \text{(defines S11 .real, S22 .real)} \]
\[ S[3].\text{imag} = S[0].\text{imag} = \text{im}s11; \text{(defines S11 .imag, S22 .imag)} \]
\[ S[2].\text{real} = S[1].\text{real} = \text{res21}; \text{(defines S12 .real, S21 .real)} \]
\[ S[2].\text{imag} = S[1].\text{imag} = \text{im}s21; \text{(defines S12 .imag, S21 .imag)} \]
Adding Noise Characteristics

Because the U2PB coaxial section is lossless, it is also noiseless; therefore, the Noise Analysis Function check box in the Code Options dialog box is not selected.

The next step is compiling and linking the C-code. See Chapter 1, Building User-Compiled Analog Models.
Creating Linear Circuit Elements
Chapter 3: Creating Nonlinear Circuit Elements

This chapter describes creating nonlinear circuit elements. Nonlinear user-defined element modeling described in this chapter is applicable to steady-state analysis only. Refer to Chapter 4, Creating Transient Circuit Elements, 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
Creating Nonlinear Circuit Elements

**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 Chapter 2, Creating Linear Circuit Elements). 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_ni 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 Figure 3-1 is used for the PNDIODE element.

![Figure 3-1. PNDIODE element model](image)

---

3-4  User-defined P-N Diode 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:
Creating Nonlinear Circuit Elements

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

```
static UserParamType
PNDIODEParms[] =
{
  {"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):

```c
#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() */
};
```
Creating Nonlinear Circuit Elements

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

```c
#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 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:

```c
#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
#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
```
Creating Nonlinear Circuit Elements

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.

```c
 /*--------------------------------------------------------------------*/
 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.

```c
static void diode_nl_iq_gc ( 
    UserInstDef *userInst, /* Changed from SIV to be consistent w/CUI */
    double *vPin,
    double *id,
    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 )
    {
        exparg = ( vd/vte < 40.0 ) ? vd/vte : 40.0;
        evd = exp(exparg);
        *id = csat * (evd - 1.0) + GMIN * vd;
        *gd = csat * evd / 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;
        }
    }

    /*
```

User-defined P-N Diode Model
3-12 User-defined P-N Diode Model

Creating Nonlinear Circuit Elements

* charge storage elements
*/  
fcpb = FC_P * VJ_P;

czero = CJ0_P * AREA_P;
if (vd < fc)
{
    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:

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

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

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

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

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

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

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.

```c
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() */
```
Creating Nonlinear Circuit Elements

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.

```c
/*--------------------------------------------------------------------*/
static boolean add_n_branch(
    UserInstDef *userInst,
    int n1,
    int n2,
    COMPLEX iNcorr)
{
    boolean status = TRUE;
    status = add_lin_n(userInst, n1, n1, iNcorr) &&
        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_nl_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, A_F_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);
    } /* analyze_ac_n() */

The next step is compiling and linking the code. Refer to “Creating the Code and Compiling the Model” on page 1-22.

Referencing Data Items
Refer to “Referencing Data Items” on page 1-43.

Displaying Error/Warning Messages
Refer to “Displaying Error/Warning Messages” on page 1-44.
Creating Nonlinear Circuit Elements
Chapter 4: Creating Transient Circuit Elements

This chapter 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:

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

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.
Creating Transient Circuit Elements

**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 Figure 4-1. Naturally, values for these components could be calculated from the `UserInstDef` parameters that are passed into the function.

```c
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;
}
```

![Figure 4-1. Element model for transient analysis](image)

---

4-2 Requirements for Creating Transient Elements
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 Figure 4-2). 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.

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.

```c
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;
}
```

Figure 4-2. 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.
User-defined P-N Diode Model

This section shows how to extend the P-N diode example created in Chapter 3 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 Figure 4-3 is used for the PNDIODE element.

![Figure 4-3. PNDIODE element model](image-url)
Defining the Transient Device

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

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

A UserTranDef structure is defined for transient.

```c
#define ANALYZE_TR_DEF_PTR analyze_tr_def_ptr
static UserTranDef analyze_tr_def_ptr =
{
    NUM_TRANSIENT_INT_NODES,  /* numIntNodes */
    NUM_TRANSIENT_CAPS, /* numCaps */
    NUM_TRANSIENT_INDS,  /* numInds */
    NUM_TRANSIENT_TLS,  /* numTlines */
    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.

```c
#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 */
};
```
Creating Transient Circuit Elements

Transient Analysis Function

The analysis routine \texttt{diode\_nl\_iq\_gc} that was written for the nonlinear model (Refer to Chapter 3, Creating Nonlinear Circuit Elements) can also be used for the transient model. Add to this the contribution of the series resistance and the model is complete. The \texttt{analyze\_tr} function that follows calls the \texttt{diode\_nl\_iq\_gc} function for the nonlinear contribution and loads them into the matrix, and then uses \texttt{add\_tr\_resistor} to include the contribution of the series resistance.

```c
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)
    {
        (void)sprintf(ErrMsg, "Error: PNDIODE: analyze\_tr(%s) -> %s", userInst->tag, pMsg);
        send\_info\_to\_scn(ErrMsg);
    }
    return status;
} /* analyze\_tr() */
```

4.6 User-defined P-N Diode Model
The next step is compiling and linking the code. Refer to Chapter 1, Building User-Compiled Analog Models.

**Referencing Data Items**
Refer to “Referencing Data Items” on page 1-43.

**Displaying Error/Warning Messages**
Refer to “Displaying Error/Warning Messages” on page 1-44.
Creating Transient Circuit Elements
Chapter 5: Custom Modeling with Symbolically-Defined Devices

This chapter 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.
Custom Modeling with Symbolically-Defined Devices

The examples in this chapter 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 chapter has the following sections:

- “Writing SDD Equations” on page 5-3 explains how to write the equations that define an SDD.
- “Adding an SDD to a Schematic” on page 5-13 describes how to add an SDD to a schematic and enter equations.
- “SDD Examples” on page 5-17 show how to use SDDs to define a wide range of nonlinear circuit components.
- “Modified Nodal Analysis” on page 5-45 is a discussion of modified nodal analysis and branch equations.
- “Error Messages” on page 5-50 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.

![Figure 5-1. The schematic symbol for a two-port SDD](image)

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 \(_i_1\) and \(_v_1\), respectively. You can rename variables to better suit the device being modeled. In text, \(v_n\) and \(i_n\) are used to refer to \(_v_n\) and \(_i_n\).

By convention, a positive port current flows into the terminal marked +.
Custom Modeling with Symbolically-Defined Devices

**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:

\[
i_k = f(v_1, v_2, \ldots, v_n)
\]

An example of an explicit equation is:

\[
I[1,0] = \frac{v_1}{50}
\]

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:

\[ f_k(v_1, v_2, ...v_n, i_1, i_2, ...i_n) = 0 \]

An example of an implicit equation is:

\[ F[1,0] = v_{bi} + i_b R_{bb} - v_b \]

This equation is part of the Gummel-Poon example.

If you want to use the current variable (\_in\_) 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” on page 5-13.
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 modal analysis, refer to the section “Modified Nodal Analysis” on page 5-45.

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” on page 5-45).

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 _in 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” on page 5-28.

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:

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

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.

A procedure for how to enter weighting functions as part of an SDD definition is in the section “Defining a Weighting Function” on page 5-16.
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:

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 \( i_1(t) \).
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” on page 5-25 and “Nonlinear Inductors” on page 5-32, where the weighting function 1 is used to implement nonlinear capacitors and inductors.
Custom Modeling with Symbolically-Defined Devices

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” on page 5-15. An example appears in “Controlling Current, Instantaneous Power” on page 5-34.

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:

\[ F[1,0] = f1(v1, i1) \]
\[ F[1,1] = f2(v1, i1) \]

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 can generate noise only for AC and S-parameter simulations. 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:

\[ I_n = 1e^{-12} + 1e^{-6}/(\text{freq}+1) \]

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 \( v_n \). The current is denoted \( i_n \). 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 can generate noise only for AC and S-parameter simulations.
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.

\[ \text{identifier} \quad \text{function} \]

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” on page 5-4.
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 $jw$). For more information on weighting functions, refer to the section "Weighing Functions" on page 5-8. For information on the procedure for adding a different weighting function to an SDD, refer to the section “Defining a Weighting Function” on page 5-16.

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[Repeater] field, type the instance name of the device.

An example of a parameter definition is shown here.

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 _cn in your SDD equation, for example, _v2 + _v1*_c1.

7. Click OK to accept the changes and dismiss the dialog box.
Custom Modeling with Symbolically-Defined Devices

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 \( jw \)). 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” on page 5-18
- “Ideal Amplifier Block” on page 5-20
- “Ideal Mixer” on page 5-23
- “Nonlinear Capacitors” on page 5-25
- “Full Model Diode, with Capacitance and Resistance” on page 5-28
- “Nonlinear Inductors” on page 5-32
- “Controlling Current, Instantaneous Power” on page 5-34
- “Gummel-Poon BJT” on page 5-36

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

Tutorials/ SDD_Examples_prj/ 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_prj/networks/Cubic.dsn

The nonlinear two-terminal resistor with constitutive relationship

\[ i(v) = \frac{v^3}{3} - v \]

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.

\[ i(1,0) = \frac{-v_1^3}{3} - v_1 \]

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 Figure 5-2.
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 1 MHz, 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_prj/ networks/ NonlinearAmp.dsn

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

\[ v_o = V_s \tanh(A v_i / V_s) \]

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 Figure 5-4 (a).

The amplifier is a two-port device, so one more equation is required to specify the constitutive relationship. In the case of Figure 5-3 (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 Figure 5-3 (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.

![Figure 5-3. Equivalent Circuit Model for an Ideal Saturating Amplifier](image)
Current through the input resistance $R_i$ can be expressed as:

$$i_1 = \left( \frac{v_i}{R_i} \right).$$

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 $i_n$ is available.

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

---

SDD Examples 5-21
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).

• $v_1$, $v_2$, and $i_2$ are assigned to variables ($v_i$, $v_o$, and $i_o$, 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 design TestAmp.dsn. DC and harmonic balance simulation results are shown in Figure 5-4.

• 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.

Figure 5-4. Simulation Results for the Ideal Saturating Amplifier
Ideal Mixer

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

Tutorials/ SDD_Examples_prj/ networks/ IdealMixer.dsn

The equivalent circuit for an ideal mixer is shown in Figure 5-5.

![Figure 5-5. Equivalent Circuit for an Ideal Mixer](image)

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:

\[ i_{rf} = \frac{v_{rf}}{Z} \]
\[ i_{lo} = \frac{v_{lo}}{Z} \]
\[ i_{if} = -\left(\frac{v_{if} - v_{rf}v_{lo}}{Z}\right) \]
These equations are voltage-controlled and can be implemented using explicit SDD equations. The SDD is shown next.

In this setup \(v_1\), \(v_2\), and \(v_3\) 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 Zs.

**Figure 5-6** 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.
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:

\[
q = \frac{Q(v)}{v}
\]

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

$$Q(v) = \int C(\dot{v}) d\dot{v} + Q_0$$

where we have explicitly included the arbitrary constant of integration $Q_0$.

If for some reason, the charge cannot be calculated, then the alternative technique presented in “Alternative Implementation of a Capacitor” on page 5-47 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” on page 5-47 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_prj/ networks/ SDD_Diode.dsn

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

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

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:

\[ Q_r(v) = -2C_o \sqrt{V_d(V_o-v)} \quad v < V_o \]

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 \( \alpha \) 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 \( \alpha V_o \) 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

\[
Q_f(v) = (v - \alpha V_o)(C_f(\alpha V_o) + C_r(\alpha V_o)(v - \alpha V_o)/2) + Q_r(\alpha V_o) \quad v \geq \alpha V_o.
\]

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

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

**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 Eqn component under Current equations. These include the variables \( \text{max\_exp} \), \( \text{max\_arg} \), the function \( \text{exp\_soft}(x) \), and the variable \( Vt \). They determine what value is multiplied by \( \text{exp\_soft} \) 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 \( \text{max\_exp} \).
  - The second equation models the charge. It uses the expressions listed in the Var Eqn 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 values are passed from TestDiode.dsn).
• 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 $v_1$ is inverse Fourier transformed into the time domain.

• The constitutive relation (in this case, $-2C_0\sqrt{V_0(V_0-v_v)}$) 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 $i_1$.

• 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 design TestDiode.dsn. This design uses the diode capacitance as the $C$ in an RC circuit. It also allows the independent adjustment of the diode bias voltage. Figure 5-7 shows the frequency response of the RC circuit as the bias voltage is varied from 1 to 2 V.

![Figure 5-7. 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 $\phi$-$i$, relationship

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

For example, the $\phi$-$i$ relationship for a linear two-terminal inductor is

$$\phi = L i$$

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

![SDD Diagram]

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

---

**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

$$\Phi(i) = \int_1^i L(i) \, di + \Phi_0$$

where we have explicitly included the arbitrary constant of integration $\Phi_0$.  

---

*SDD Examples 5-33*
Controlling Current, Instantaneous Power

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

    Tutorials/ SDD_Examples_prj/ networks/ RemCC.dsn

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” on page 5-10 and “Defining a Controlling Current” on page 5-15.

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

![Circuit diagram](image)

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:

1. 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.

   ```plaintext
   SDD2E
   SDDNP1
   [1,0]=0
   F[2,0]=v2 + _v1+c1
   C[1]="Vdss"
   ```
• 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* -cl, 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:

\[ v2 = -v1*cl \]

and by using an implicit equation, the equation

\[ v2 + v1*cl \]

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.

![Graph of Vdd and Vpow](image)
Gummel-Poon BJT

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

    Tutorials/ SDD_Examples_prj/ networks/ GumPoon.dsn

Figure 5-8 shows the equivalent circuit model for the Gummel-Poon bipolar junction transistor (BJT). The associated current and capacitance equations follow.

![Figure 5-8. Equivalent Circuit Model for the Gummel-Poon BJT](image-url)
Current Equations

\[ i_c = \frac{J_s}{Q_b} \left( e^\left( \frac{V_{bei}}{N_f V_t} \right) - e^\left( \frac{V_{bci}}{N_r V_t} \right) \right) - \frac{J_s}{B_r} \left( e^\left( \frac{V_{bci}}{N_r V_t} \right) - 1 \right) - J_{lc} \left( e^\left( \frac{V_{bci}}{N_r V_t} \right) - 1 \right) \]

\[ i_c = \frac{J_s}{B_f} \left( e^\left( \frac{V_{bei}}{N_f V_t} \right) - 1 \right) - \frac{J_s}{B_r} \left( e^\left( \frac{V_{bci}}{N_r V_t} \right) - 1 \right) - J_{le} \left( e^\left( \frac{V_{bci}}{N_r V_t} \right) - 1 \right) + J_{lc} \left( e^\left( \frac{V_{bci}}{N_r V_t} \right) - 1 \right) \]

\[ R_{bb} = 3 \cdot (rb - rbm) \cdot \left( \tan(Z) - Z \right) / \left( Z \cdot \tan(Z)^2 \right) + rbm \]

where

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

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

\[ Q = \frac{J_s}{B_f} \left( e^\left( \frac{V_{bei}}{N_f V_t} \right) - 1 \right) + \frac{J_s}{B_r} \left( e^\left( \frac{V_{bci}}{N_r V_t} \right) - 1 \right) \]
Capacitance Equations

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

\[ C = \frac{T_{rs}}{N_r V_t} e^{\frac{V_{bci}}{V_T}} \left( 1 + X_{jc} C_{jc} \left( 1 - \frac{V_{bci}}{V_{jc}} \right)^{-M_{jc}} \right) \]

\[ C_{be} = \frac{\partial}{\partial V_{bei}} \left( T_{rf} \frac{(V_{bei})}{Q_b} \left( e^{\frac{V_{bei}}{N_f V_t}} - 1 \right) + C_{je} \left( 1 - \frac{V_{bei}}{V_{je}} \right)^{-M_{je}} \right) \]

where

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

\[ I_f = J_s \left( e^{\frac{V_{bei}}{N_f V_t}} - 1 \right) \]

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:

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

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

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

$$i_e = -i_b(v_{bei}, v_{bci}) - i_c(v_{bei}, v_{bci}) - \frac{d}{dt} Q_{be}$$

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 $i_c$:

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

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

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

$$i_e = i_b(v_{bei}, v_{bci}) - i_c(v_{bei}, v_{bci}) - \frac{d}{dt} Q_{be}$$
Custom Modeling with Symbolically-Defined Devices

SDD Implementation

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

Tutorials/SDD_Examples_prj/networks/GumPoon.dsn

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

transistor variables:

\[
egin{align*}
 v_{be} &= v_{b} - v_{a} \\
 v_{be} &= v_{b} - v_{e} \\
 v_{bc} &= v_{b} - v_{c} \\
 v_{bc} &= v_{b} - v_{c'} \\
 I_b &= I_b_1 = I_b_2 \\
 I_c &= I_c_1 = I_c_2 \\
 V_{ce} &= V_{ce} = V_{ce} \\
 V_{ce}/2 &= V_{ce}/2 = V_{ce}/2 \\
 \end{align*}
\]

supporting equations follow
VAR

VAR1

& temperature variables and equations

$V_{ce} = V_{cc}$

$B_rT = V_{cc}$

$E(t) = e^{0.7T2}\cdot V_{ce} - 2/\ln(0.7T2)$

$B_f T = 6.9$

$e_{same}=9$

$E_{T} = E_{T} - 2000 \cdot \exp\left(\frac{E_{T}}{E_{T}}\right) \cdot \exp\left(\frac{V_{T}}{E_{T}}\right) \cdot \exp\left(\frac{V_{T}}{E_{T}}\right)$

$T_{zero} = 0.7T2 \cdot 1.5$

$T_{e} = T_{e}$

& non-linear base resistance

$R_b = \frac{1}{1 + \sqrt{1 + 4\cdot R_b \cdot (2\cdot J_b + 1)}}$,

$A_s = \frac{1}{2\cdot \tan(2) - 2}$,

$A_{s1} = \frac{1}{2\cdot \tan(2) + 2}$

& base-collector charges

$q_{b2} = q_{b1} + q_{b1} \cdot (1 - m) / (1 - n)

$q_{b2} = q_{b1}

& base-emitter charge

$q_{e2} = q_{e1} + q_{e1} \cdot (1 - m) / (1 - n)

$q_{e2} = q_{e1}

& collector current equations

$I_C = I_C(1 - V_{bb} / V_{bb})$,

$I_C = I_C(1 - V_{bb} / V_{bb})$,

$I_C = I_C(1 - V_{bb} / V_{bb})$

& intermediate variables for collector current

$Q_1 = I_C(1 - V_{bb} / V_{bb})$

$Q_2 = I_C(1 - V_{bb} / V_{bb})$

$Q_3 = I_C(1 - V_{bb} / V_{bb})$

& useful general equations

$V_{bb} = \exp\left(\frac{V_{bb}}{E_{T}}\right)$,

$\exp\left(\frac{V_{bb}}{E_{T}}\right) = 1$

$\exp(\text{max}) = \text{max}(\text{exp})$
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 $C_{be}$ since the first term is a partial derivative, the integration and partial derivative effectively cancel.
  - The integration is simplified for the first term in $C_{bc}$ 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” on page 5-28.
- 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 $V_{je}$ and $V_{jc}$ 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
  \[ Q(v) = \int C(v) dv + Q_0 \]
  where \( Q_0 \) is the arbitrary constant of integration.
Custom Modeling with Symbolically-Defined Devices

• A **current-controlled nonlinear inductor** is described by its $\phi$-$i$ relation

  \[ \phi = \Phi(i). \]

• A **two-terminal current-controlled nonlinear inductor** $\phi = \Phi(i)$ is implemented by

  \[
  I[1,0] = _v1 \\
  I[1,1] = -\text{phi}_1(i1)
  \]

• 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

  \[ \Phi(i) = \int L(i) di + \Phi_0 \]

  where $\Phi_0$ 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 Kirchoff’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 \( \mathbf{i} \) be the vector of branch currents. Then KCL can be expressed by the equation

\[
A\mathbf{i} = 0
\]

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

\[
\mathbf{i} = g(\mathbf{v})
\]

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

\[
\mathbf{G}(\mathbf{v}) = 0
\]

where \( \mathbf{G} \) is a map from \( \mathbb{R}^n \) to \( \mathbb{R}^n \) defined by \( \mathbf{G}(\mathbf{v}) = A\mathbf{g}(\mathbf{v}) \).
Custom Modeling with Symbolically-Defined Devices

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

\[ \mathbf{G}(\mathbf{v}, \mathbf{i}_b) = 0 \]

where \( \mathbf{i}_b \) is the vector of the \( n_b \) branch currents of the non-voltage-controlled devices and \( \mathbf{G} \) is a map from \( \mathbb{R}^{n+n_1} \) to \( \mathbb{R}^n \). Since there are now \( n \) equations in \( n+n_b \) 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

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

The resulting augmented nodal equations are the modified nodal analysis equations

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

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

\[
dv/dt = \frac{dv}{dt}.
\]

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

\[
dv/dt = j2\pi fv.
\]
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 \( \frac{dv}{dt} \). This high sensitivity can cause convergence difficulties for the system. To eliminate the problem, scale by a nominal frequency value of 1 GHz.

\[
f_{\text{nom}} = 1\text{GHz} \\
i = C(v) f_{\text{nom}} \frac{dv}{dt}
\]

\[
\frac{dv}{dt} = \frac{1}{f_{\text{nom}}} \frac{dv}{dt}.
\]

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

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

\[
I[1,0] = C(v) f_{\text{nom}} \frac{dv}{dt}
\]

\[
F[2,0] = -\frac{dv}{dt}
\]

\[
F[2,1] = v/f_{\text{nom}}
\]

and these VAR equations:

\[
v = _v1
\]

\[
\frac{dv}{dt} = _v2
\]

\[
f_{\text{nom}} = 1\text{GHz}
\]

\[
C(v) = 1\text{pf*exp(v)}
\]

This SDD can be found in Examples/Tutorials/SDD_Examples_prj as SDD_cap.dsn. An alternate implementation, SDD_cap2.dsn, can also be found in the project.
Alternative Implementation of a Capacitor

\[ F[1,0] = C(v) \times f_{nom} \times \frac{dv}{dt} \]
\[ F[2,0] = -\frac{dv}{dt} \]
\[ F[2,1] = \omega_f \times f_{nom} \]

VAR1
\[ v = v_1 \]
\[ \frac{dv}{dt} = v_2 \]
\[ f_{nom} = 1 \text{ GHz} \]

VAR2
\[ C(v) = 1 \text{ pf} \times \exp(v) \]
Custom Modeling with Symbolically-Defined Devices

**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.)
Chapter 6: 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 allows 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 Chapter 5, Custom Modeling with Symbolically-Defined Devices) 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 chapter, you should be familiar with the SDD. This chapter 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 Chapter 5, Custom Modeling with Symbolically-Defined Devices.

### 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 ($0.5/T_{\text{step}}$). 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 \(\text{freq1}\) and \(\text{freq2}\) are fundamental frequencies.

For example, consider a two-fundamental simulation, with fundamental frequencies \(\text{freq1}\) and \(\text{freq2}\) defined as 1 GHz 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 Table 6-1 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.
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.

Table 6-1. Example Signals for Spectrum around $f_k$

<table>
<thead>
<tr>
<th>#</th>
<th>Formula</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$V_k=1$</td>
<td>Constant cosine $\cos(2\pi f_k t)$</td>
</tr>
<tr>
<td>2</td>
<td>$V_k=\exp(j\pi/2)$ or polar(1,-90) or $-j$</td>
<td>Constant sine $\sin(2\pi f_k t)$</td>
</tr>
<tr>
<td>3</td>
<td>$V_k=A\exp(j(2\pi f_m t+B))$</td>
<td>One tone (SSB) $A\cos(2\pi(f_k+f_m) t+B)$</td>
</tr>
<tr>
<td>4</td>
<td>$V_k=A\exp(jB)$; freq=1.1 GHz$^1$</td>
<td>Same as (3) (assuming $f_k + f_m = 1.1$ GHz)</td>
</tr>
<tr>
<td>5</td>
<td>$V_k=2\cos(2\pi f_m t)$</td>
<td>Two tone (AM suppressed carrier)</td>
</tr>
<tr>
<td>6</td>
<td>$V_k=\exp(j2\pi f_m t) + \exp(-j2\pi f_m t)$</td>
<td>Same as (5)</td>
</tr>
<tr>
<td>7</td>
<td>$V_k=pulse(time,...); freq=f_k+f_m$</td>
<td>Pulsed RF at a frequency of $f_k + f_m$</td>
</tr>
<tr>
<td>8</td>
<td>$V_k=-step(time - delay)$</td>
<td>A negative cosine wave, gated on at $t=delay$</td>
</tr>
<tr>
<td>9</td>
<td>$V_k = (\text{vreal}(time)+j\text{vimag}(time))\exp(j2\pi f_m t)$</td>
<td>I/Q modulated source centered at $f_k+f_m$ (\text{vreal()}, \text{vimag()} user-defined functions)</td>
</tr>
<tr>
<td>10</td>
<td>$V_k=(1 + vr1) * \exp(j2\pi vr2)$</td>
<td>Amplitude and noise modulated source at $f_k$ (vr1, vr2 user-defined randtime functions)</td>
</tr>
<tr>
<td>11</td>
<td>$V_k=\exp(j2\pi (-f_0 + a0*time/2)t)$</td>
<td>Chirped FM signal starting at $f_k-f_0$ rate $a_0$</td>
</tr>
</tbody>
</table>

1. freq is defined in the paragraph below.
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 Table 6-1, 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” on page 6-23. For examples of FDDs developed into models, refer to the section “FDD Examples” on page 6-28.

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 Table 6-2, and used in conjunction with equations, which are described in the section “Defining Constitutive Relationships with Equations” on page 6-8. 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” on page 6-10. For a description of frequency indices from a harmonic balance analysis, refer to the section “Signal Models and Sources” on page 6-2.
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 Table 6-2. These functions have a delay parameter. For example, to find the value of the port 2 voltage at the \([2,-1]\) intermod frequency 10 \(\mu\)sec ago, use \_sv_d(2, 10us, 2,- 1).

### Table 6-2. Functions for Accessing Spectral Ports and Currents

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_sv(port, findex) †</td>
<td>Returns the spectral voltage for the specified port at the specified frequency index.</td>
</tr>
<tr>
<td>_si(port, findex)</td>
<td>Returns the spectral current for the specified port at the specified frequency index.</td>
</tr>
<tr>
<td>_sv_d(port, delay, findex)</td>
<td>Returns a past value of spectral voltage for the specified port and time delay at the specified frequency index.</td>
</tr>
<tr>
<td>_si_d(port, delay, findex)</td>
<td>Returns a past value of spectral current for the specified port and time delay at the specified frequency index.</td>
</tr>
</tbody>
</table>

†\(findex\) 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

\[
\text{fdd\_delay} = \begin{cases} 
\text{max\_delay} & \text{if } (\text{time} = 0) \\
\text{variable\_delay} & \text{else}
\end{cases}
\]

where \text{max\_delay} is some reasonable value that the variable\_delay is known to never exceed.
Custom Modeling with Frequency-Domain Defined Devices

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:

\[ v[2,0] = \begin{cases} 
0.0 & \text{if } (\text{time} = 0) \\
\_sv\_d(2,\text{timestep}) + \_sv(1,0) & \text{else}
\end{cases} \]

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.

\[ \text{next\_state} = \_sv\_d(2,\text{timestep}) + \_sv(1,0) \]
\[ v[2,0] = \begin{cases} 
0.0 + \text{next\_state} \times 1.0e-100 & \text{if } (\text{time} = 0) \\
\text{next\_state} & \text{else}
\end{cases} \]

**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:

\[ i[\text{port}, \text{findex}] = f(\_sv(), \_sv\_d(), \_si(), \_si\_d()) \]
\[ v[\text{port}, \text{findex}] = f(\_sv(), \_sv\_d(), \_si(), \_si\_d()) \]

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 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” on page 6-2 and “Specifying Carriers with the Freq Parameter” on page 6-10.

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” on page 5-45, for more detail.
While the SDD has truly implicit equations of the form \( f(v_1, ..., v_n, i_1, ..., i_n) = 0 \) 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

\[
v[2, all \text{other harmonic indices}] = 0.0.
\]

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., \( v[2,0] = v_{calc} \)), 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” on page 6-24.

**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.
Custom Modeling with Frequency-Domain Defined Devices

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. 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 them 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. For example, if an FDD has defined freq[3]=800 MHz, then _sv(1,-3,3) specifies the port 1 spectral voltage at the envelope closest to 2.4GHz. 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. Note that _sv(1,3,3) would be unaffected by the carrier frequency parameters but 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" on page 6-5) 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] = 500MHz and freq[2]=500.1MHz, 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 1msec, 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" on page 6-31.

For a procedure on how to add freq parameters to an FDD, refer to the section "Defining Frequency Parameters" on page 6-25.
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 chapter, 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:

```plaintext
parameters VPEAK=1 V DUTYCYCLE=30 FREQ=1 GHz HARMONICS=16

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 1msec (+/-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*freq1 + 2*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: CMP2 _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: CMP3 0 _NET00005 C=1/ (2*pi*Kv)
vco_harm: CMP1 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 \texttt{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 \texttt{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 \texttt{harm} index can also be used to change which spectral input is used for each spectral output. An example is \(i[2,1] = \texttt{sv}(2,\_\text{harm})/50\), which adds a 50 ohm load at all the harmonics of \texttt{fund1}, including DC. Note that this example is to illustrate the capability, it is inefficient compared to using a resistor. Another example is \(i[2,1] = \texttt{sv}(1,0,\_\text{harm})/\text{mag}(\texttt{sv}(1,0,1)+\text{tinyreal})/\text{Rout}\), which outputs an entire \texttt{fund1} spectrum at port 2, based on the port 1 \texttt{fund2} spectrum, and limits each spectral component by the \texttt{fund2} fundamental magnitude.

Note there is no capability in the FDD to allow automatic outputting of all spectral tones. The \texttt{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" on page 6-31.
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 \texttt{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, \texttt{list(1, \_02\_j\_01)}.

```plaintext
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]=.001
; This capacitor performs the integration function
C:CM\_P 3 0 _NET00005 C=1/ (2\_pi\_Kv)
; This switch resets the integrating capacitor voltage to 0 at time =0
ResetSwitch:CM\_P 4 0 _NET00005
; This FDD is a programmable harmonic current source with a phase modulation input
FDD:CM\_P 2 _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 =-\_dbmto\_a(Pdbm, Rout)*exp(j*phaserad(P))
b =1000* _sv\_d(1,Delay,0)
R:CM\_P 5 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.

```vhdl
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:CM P1 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” on page 6-18.

Three built-in functions have been defined to provide access to trigger information, they are described in Table 6-3. 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.

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

Another function is available to detect threshold crossings and to generate the proper trigger expression values, which is shown in Table 6-4. 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.

Table 6-4. Function to generate a trigger event

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_xcross(P, Vthresh, direction)</td>
<td>Returns 0 if no threshold crossing occurred, otherwise returns its relative time, a value between ([-1 \text{ and } -2) \times \text{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 negate number a negative edge; a direction number of 0 implies either positive or negative edge. No hysteresis exists.</td>
</tr>
</tbody>
</table>

For a procedure on how to add trigger parameters to an FDD, refer to the section “Defining Triggers” on page 6-26.

**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 \(\text{bin()}\) 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 \(\text{bin}(101)\) 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” on page 6-34.

For a procedure on how to add clock enable parameters to an FDD, refer to the section “Defining Clock Enables” on page 6-27.
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 Table 6-5.

The \_sv\_e() function is very similar to \_sv\_d(), which is described in Table 6-2. 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_sv_e(P,N,indices)</td>
<td>Return the port P spectral voltage envelope at the last trigger N time</td>
</tr>
<tr>
<td>_si_e(P,N,indices)</td>
<td>Return the port P spectral current envelope at the last trigger N time</td>
</tr>
<tr>
<td>_sv_bb(P,N)</td>
<td>Return the total, real voltage of port P at the last trigger N time</td>
</tr>
<tr>
<td>_si_bb(P,N)</td>
<td>Return the total, real current entering port P at the last trigger N time</td>
</tr>
</tbody>
</table>


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

$$V_k(t) \cdot \exp(j \cdot 2 \cdot \pi \cdot f_c \cdot t)$$

then

$$i_{[2,1]} = _sv_d(1,1 \mu s,1)$$

generates a current equal to

$$V_k(t-1 \mu s) \cdot \exp(j \cdot 2 \cdot \pi \cdot f_c \cdot t)$$

To generate a true coherent delay with the FDD, you would have to modify the equation to

$$i_{[2,1]} = _sv_d(1,1 \mu s,1) \cdot \exp(-j \cdot 2 \cdot \pi \cdot f_c \cdot 1 \mu s)$$

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 Table 6-6. 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.

Table 6-6. Miscellaneous FDD Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_divn(T,N,N0)</td>
<td>Returns the value of a counter, clocked every time trigger T occurs, decrementing from N to 0. N0 is initial time = 0 value.</td>
</tr>
<tr>
<td>_lfsr(T, seed, taps)</td>
<td>Returns the value of a linear feedback 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.</td>
</tr>
<tr>
<td>_shift_reg(T, M, N, In)</td>
<td>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</td>
</tr>
</tbody>
</table>
Defining Input and Output Impedances

With the SDD, it is very straightforward to include the input and output resistances in the basic equations. For example, \( i[1] = \frac{v1}{50} \) 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, \( i[1,1,0] = \frac{sv(1,1,0)}{50} \) 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 the Circuit Components manuals.
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” on page 6-8.

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” on page 6-10.

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 nth 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” on page 6-16.

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 Nth bit corresponds to whether this port should be enabled by the Nth trigger. For more information, refer to the section “Output Clock Enables” on page 6-17.

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” on page 6-29
- “Mixer” on page 6-31
- “Sample and Hold” on page 6-34

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

  Tutorials/FDD_Examples_prj/networks
IQ Modulator

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

Tutorials/FDD_Examples_prj/networks/IQ_modulator.dsn

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.
Custom Modeling with Frequency-Domain Defined Devices

The design IQmodTest.dsn 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.
Mixer

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

```
Tutorials/FDD_Examples_prj/networks/FDDmixer.dsn
```

This is a simple, ideal mixer. It models upconversion, downconversion, LO leakage, RF leakage, and conversion gain compression with increasing LO amplitude.
Custom Modeling with Frequency-Domain Defined Devices

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 designs FDDmixerTest.dsn and FDDmixerTestEnv.dsn 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_prj/ networks/ SampleHold.dsn

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” on page 6-16.

- 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” on page 6-17.
Custom Modeling with Frequency-Domain Defined Devices
Chapter 7: Building Signal Processing Models

Agilent 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. Agilent Ptolemy includes the Model Development Kit, a feature that allows you to create, compile, and link your models into Agilent 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 Agilent Ptolemy Preprocessor Language, described in Chapter 8, Writing Component Models.

---

**Note**  In versions prior to ADS 2001, there was an option of building Signal Processing models using the graphical user interface (GUI) method. However, due to the limitations of this method in building Signal Processing models, this option has been dropped for Signal Processing models (it is still available for Analog/RF System models). Refer to the section “Advanced Model Building Functions” on page 7-2 to see some of the limitations with the dropped GUI method. None of the listed capabilities were available in the GUI method.

User-defined models (and their associated files) that you created with the older GUI method are still usable with the current command-line method. Follow the instructions in this chapter to learn how to use these models (for example, moving files to the proper directories, etc.).

---

**Note**  Definitions for UC Berkeley Ptolemy terms, such as star and particle, are given in the Agilent Ptolemy Simulation documentation.
Advanced Model Building Functions

This chapter describes processes that provide you with the full features available in Agilent 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 (in hpeesoflang).

This chapter 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 manual for the platform you are using:

- For UNIX, see Installation on UNIX Systems > “Before You Begin” > “Check the System Requirements”.
- For the PC, see Installation on PC Systems > “Before You Begin” > “Check the System Requirements”.

Note: For HP-UX, the requirements include the C compiler in cases where external C files or routines will be compiled and linked in to the C++ Agilent Ptolemy model. For Solaris and Windows, an ANSI C compiler is included with the C++ compiler and will work for the ADS Analog Model Development Kit.

Although the Microsoft Visual C++ installation prompts you to create certain environment variables (in the Setup Environment Variables dialog box), the default setting of the Register Environment Variables option is off. You must select this option to create the variables. If Visual C++ was previously installed without setting the variables, you must reinstall and select the aforementioned option.

The variables are written to the file VCVARS32.BAT in your VC6 bin directory.
Creating a Simple Model Library

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, or %HPEESOF_DIR%/bin under Windows, directory to your PATH.

In the Windows environment, we recommend using the “Ptolemy Modelbuilder Shell” window for modelbuilding. However, these variables can also be set from an MS-DOS window using the Set command.

To activate the “Ptolemy Modelbuilder Shell” window select: Start &gt; Programs &gt; Advanced Design System 2002C &gt; Tools &gt; Ptolemy Modelbuilder Shell from the Windows main screen, as shown in the following illustration.
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.

Set Up the Area to Build Models

A model build area can contain any number of libraries and star libraries. Each star library can contain many stars. In your home directory, we'll create a model build area with a single star library for this tutorial:

In UNIX:

```
cd hpeesofmb
```

and in Windows:

```
cd c:\users\default\hpeesofmb
```

The hpeesofmb command will create a directory called hptolemy and copy some files there. Inside the hptolemy 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 code for your libraries and stars will go.
Building Signal Processing Models

**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.

**Note** If you have a work area that was generated using the ADS Model Builder prior to ADS 2002 release, you must clean-up and update your makefiles. To do this, do the following:

```bash
cd hptolemy
rm mk/*
cd ..
hpeesofmb -clean
```

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:**

```bash
cd hptolemy/src
cp $HPEESOF\_DIR/doc/sp_items/SDFSin.pl SDFMySin.pl
vi SDFMySin.pl
```

**In Windows:**

```bash
cd hptolemy\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:

```c
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. 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

```c
PL_SRCS = SDFMySin.pl
```

If you are using the SPARC Compiler version 5, you will also need to uncomment the indicated line in the make-defs. If you’re not sure which version of the SPARC Compiler you are using, type `cc --version`.

**Build the Shared Library**

To build a shared library and install it into your build area, you need to run `hpeesofmake`. Run this command from the hptolemy directory:

In UNIX:

```bash
cd ~/hptolemy
hpeesofmake “debug=1”
```

and in Windows:

```bash
cd c:\users\default\hptolemy (where c: is the drive letter where you have ADS installed)
hpeesofmake “debug=1”
```

**Note** The install target of hpeesofmake has been changed in ADS 2002. To build a shared library, you no longer need to use the install target.
Due to changes in ADS 2002, if you have any shared libraries that were generated using the hpeesofmake install command prior to the ADS 2002 release (such as ADS 2001 or ADS 1.5), you may have to regenerate them using the ADS 2002 hpeesofmake command. To do this, refer to “Set Up the Area to Build Models” on page 7-5, for details on how to run hpeesofmake -clean. If you fail to do this, your older shared libraries will not work in ADS 2002.

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” on page 7-16 for more details.

Building the shared library will take some time. If you do a listing of the hptolemy directory, you’ll see two new directories, lib.arch and obj.arch, where arch is an abbreviation for your architecture.

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 building 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.
Build the AEL, Default Bitmaps, and Default Symbols

To use your star in the Signal Processing schematic, you must generate the associated AEL, bitmap, and symbol. Create them with the following commands. These should be run from the hptolemy directory:

```bash
hpeesofmake ael
hpeesofmake bitmap
hpeesofmake symbol
```

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.

Note  Due to changes in ADS 2001, if you have any AEL files that were generated using `hpeesofmake ael` command prior to the ADS 2001 release (such as ADS 1.5 or ADS 1.3), you must regenerate them using the ADS 2001 `hpeesofmake ael` command. If you do not do this, your older models will not work with ADS 2001.

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 ordinary but serve as templates for you to edit further. They will be the right size and have the appropriate number and type of pins. Symbols can be edited from ADS. Open them as you would any other schematic. 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 HPTOLEMY_MODEL_PATH environment variable to point to your model build area. The simulator uses this variable to find your libraries, symbols, bitmaps, and AEL. The variable is a colon delimited path in UNIX and a semicolon delimited path in Windows.

For this example, set it to $HOME/hptolemy in UNIX or c: \users\default\hptolemy in Windows. Now start ADS. You'll see your star on the palette on the left under wherever you set the location field above.

You can rebuild your 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 HPTOLEMY_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

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,
Building Signal Processing Models

    INCLUDEPATH += /libtree/headers
    SRCS += myutilities.c moreutilities.cxx
    OBJ S += /libtree/objs/tree$(OBJ SUFFIX)
    #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
    #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 hptolemy/src, you must set a particular make-defs variable to 1. Each directory has a corresponding variable according to the table:
At a minimum, you will need to set the variable corresponding to the domain 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 Agilent Ptolemy stars in the doc/sp_items directory. These are the stars for which you can click the C++ Code button in the online Signal Processing Components manual. 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.

<table>
<thead>
<tr>
<th>Directory</th>
<th>make-defs Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>hptolemy-kernel/compat</td>
<td>(always included)</td>
</tr>
<tr>
<td>hptolemy-kernel/kernel</td>
<td>KERNEL</td>
</tr>
<tr>
<td>numeric/kernel</td>
<td>SDFKERNEL</td>
</tr>
<tr>
<td>numeric/base/stars</td>
<td>SDFSTARS</td>
</tr>
<tr>
<td>numeric/dsp/stars</td>
<td>SDFDSP</td>
</tr>
<tr>
<td>numeric/libptdsp</td>
<td>PTDSP</td>
</tr>
<tr>
<td>timed/kernel</td>
<td>TSDFKERNEL</td>
</tr>
<tr>
<td>timed/base/stars</td>
<td>TSDFSTARS</td>
</tr>
<tr>
<td>matrix/base/stars</td>
<td>SDFMATRIX</td>
</tr>
<tr>
<td>fixpt-analysis/base/stars</td>
<td>SDFFIX</td>
</tr>
<tr>
<td>controls-displays/tcltk/ptklib</td>
<td>PTK</td>
</tr>
<tr>
<td>controls-displays/tcltk/stars</td>
<td>SDFTK</td>
</tr>
<tr>
<td>instruments/stars</td>
<td>SDFINSTKERNEL</td>
</tr>
</tbody>
</table>
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 project 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, an environment variable must be set so that the simulator can find all its shared libraries. When you ran hpeesofmb above, a script called mbsetvars was created in the bin directory. Under UNIX, you will have to evaluate its output to set the appropriate variable. Assuming you're building models in your home directory, type:

```
cd ~/hptolemy/bin
eval `./mbsetvars`
```

Under Windows, the script is a .bat file so you can run it directly:

```
cd c:\users\default\hptolemy\bin
mbsetvars
```

The mbsetvars script has the path to your model build area encoded in it, so you should not use the script from a different model build area. If you move the model build area, regenerate the script by removing it and then running the hpeesofmb command again.

Now you should be able to run your simulation from the command line by moving to your project'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++ Debugger from the command line with the msdev command. If you start it from the Start menu it will not work because it will not inherit the environment variables set by mbsetvars.

First, choose File > Open to open the bin\hpeesofsim.exe file from the ADS installation directory.

Second, choose Project > Settings, and click on the Debug tab. In the General category, set the Working directory to the data directory of your project directory and the Program arguments to ..\netlist.log. The Executable for debug session will already be set to hpeesofsim.exe. Then, in the Additional DLLs category, load the DLLs that you built from your model library's lib.win32 directory.

Now you will be able to load your code, set break points in it, and run simulations from within the Visual C++ debugger.

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 HP-UX, use the Wildebeest debugger, wdb, v1.1 or higher. (Wildebeest may be freely downloaded from http://devresource.hp.com, then search for Wildebeest.) On Solaris, use the regular debugger, dbx.

The Wildebeest debugger has the ability to load the symbol tables from your shared libraries with the shared command. After loading the libraries, you will be able to set breakpoints in your code. Run the simulator in your project's data directory with an argument of ../netlist.log.

With dbx, and if symbol table loading is not working under wdb, set a breakpoint on the hptolemy_simulate function, and run the program as described above. After this break point is reached, all libraries (including your own) will be loaded, and you will be able to set breakpoints in your code and continue execution.
Building Signal Processing Models

Platform-Specific Issues

HP-UX 11.x

aCC requires parentheses when the left-hand side of an equality contains the % operator. For example, the code:

```
output%0 = input%0
```
must be written like this:

```
(output%0) = input%0
```

If you are compiling under HP-UX 11.x, you may get an error (future) message regarding the redefinition of a macro. You can safely ignore these messages since they do not affect the quality of your compiled code. The following is an example of this type of error message:

```
Error (future) 129: "/usr/include/math.h", line 188 # Redefinition of macro 'INFINITY' differs from previous definition at ...
```

Windows

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.
Chapter 8: Writing Component Models

As described in Chapter 7, Building Signal Processing Models, you can build your own component models to supplement the large libraries included with Agilent Ptolemy. This chapter describes how to write the body of these models, and includes:

- “Writing C++ Code for Stars” on page 8-28
- “Writing Timed Components” on page 8-41
- “Using the Agilent Ptolemy Preprocessor Language” on page 8-1

Using the Agilent Ptolemy Preprocessor Language

Since the stars in Agilent 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 Agilent Ptolemy preprocessor language makes this easier.

The Agilent Ptolemy preprocessor was created to make it easier to write and document star class definitions to run under Agilent Ptolemy. Instead of writing all the class definitions and initialization code required for an Agilent 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.xlm 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 Figure 8-1. 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 {
      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;
}

Figure 8-1. SDFRect.pl File

Only one type of declaration may appear at the top level of an Agilent 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 Agilent 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

Table 8-1 provides an alphabetical list of the items that can appear in a defstar directive, including a summary of directives.

Table 8-1. Summary of Items Used to Define a Star

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Summary</th>
<th>Required</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>acknowledge</td>
<td>The names of other contributors to the star</td>
<td>No</td>
<td>8-9</td>
</tr>
<tr>
<td>attributes</td>
<td>Attributes for the star</td>
<td>No</td>
<td>8-9</td>
</tr>
<tr>
<td>attributes</td>
<td>Attributes for PortHoles</td>
<td>No</td>
<td>8-9</td>
</tr>
<tr>
<td>attributes</td>
<td>Attributes for the States</td>
<td>No</td>
<td>8-10</td>
</tr>
<tr>
<td>author*</td>
<td>The name(s) of the author(s)</td>
<td>No</td>
<td>8-8</td>
</tr>
<tr>
<td>begin</td>
<td>C++ code to execute at start time, after the schedule setup</td>
<td>No</td>
<td>8-24</td>
</tr>
<tr>
<td>ccinclude</td>
<td>Specify other files to include in the .CC file</td>
<td>No</td>
<td>8-25</td>
</tr>
<tr>
<td>code</td>
<td>C++ code to include in the .CC file outside the class definition</td>
<td>No</td>
<td>8-25</td>
</tr>
<tr>
<td>conscalls</td>
<td>Define constructor calls for members of the star class</td>
<td>No</td>
<td>8-23</td>
</tr>
<tr>
<td>constructor</td>
<td>C++ code to include in the constructor for the star</td>
<td>No</td>
<td>8-22</td>
</tr>
<tr>
<td>copyright</td>
<td>Copyright information to include in the generated code</td>
<td>No</td>
<td>8-10</td>
</tr>
<tr>
<td>derived</td>
<td>Alternate form of derivedfrom</td>
<td>No</td>
<td>8-7</td>
</tr>
<tr>
<td>derivedfrom</td>
<td>The base class, which may also be a star</td>
<td>No</td>
<td>8-7</td>
</tr>
<tr>
<td>desc</td>
<td>Alternate form of descriptor</td>
<td>No</td>
<td>8-8</td>
</tr>
<tr>
<td>descriptor*</td>
<td>A short summary of the functionality of star</td>
<td>No</td>
<td>8-8</td>
</tr>
<tr>
<td>destructor</td>
<td>C++ code to include in the destructor for the star</td>
<td>No</td>
<td>8-23</td>
</tr>
<tr>
<td>domain*</td>
<td>The domain and the prefix of the name of a class</td>
<td>Yes</td>
<td>8-7</td>
</tr>
<tr>
<td>explanation</td>
<td>Full documentation optionally using troff, eqn, and tbl formats</td>
<td>No</td>
<td>8-11</td>
</tr>
<tr>
<td>footnotes</td>
<td>Optional field to include some footnotes</td>
<td>No</td>
<td>8-12</td>
</tr>
<tr>
<td>go*</td>
<td>C++ code to execute when the star fires</td>
<td>No</td>
<td>8-24</td>
</tr>
<tr>
<td>header</td>
<td>C++ code to include in the .h file, before the class definition</td>
<td>No</td>
<td>8-26</td>
</tr>
<tr>
<td>hinclude</td>
<td>Specify other files to include in the .h file</td>
<td>No</td>
<td>8-25</td>
</tr>
<tr>
<td>htmldoc</td>
<td>Full documentation optionally using troff, eqn, and tbl formats</td>
<td>No</td>
<td>8-14</td>
</tr>
<tr>
<td>inmulti</td>
<td>Define a set of inputs</td>
<td>No</td>
<td>8-21</td>
</tr>
<tr>
<td>input*</td>
<td>Define an input to the star</td>
<td>No</td>
<td>8-21</td>
</tr>
</tbody>
</table>

* Indicates a minimum set of the most useful items.
Table 8-1. Summary of Items Used to Define a Star (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Summary</th>
<th>Required</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Component library (palette) name where user will find the star</td>
<td>No</td>
<td>8-11</td>
</tr>
<tr>
<td>method</td>
<td>Define a member function for the star class</td>
<td>No</td>
<td>8-26</td>
</tr>
<tr>
<td>name*</td>
<td>The name of the star and the root of the name of the class</td>
<td>Yes</td>
<td>8-7</td>
</tr>
<tr>
<td>outmulti</td>
<td>Define a set of outputs</td>
<td>No</td>
<td>8-21</td>
</tr>
<tr>
<td>output*</td>
<td>Define an output from the star</td>
<td>No</td>
<td>8-21</td>
</tr>
<tr>
<td>private</td>
<td>Define private data members of the star class</td>
<td>No</td>
<td>8-25</td>
</tr>
<tr>
<td>protected</td>
<td>Define protected data members of the star class</td>
<td>No</td>
<td>8-25</td>
</tr>
<tr>
<td>public</td>
<td>Define public data members of the star class</td>
<td>No</td>
<td>8-25</td>
</tr>
<tr>
<td>range</td>
<td>Define range of the parameter</td>
<td>No</td>
<td>8-12</td>
</tr>
<tr>
<td>setup*</td>
<td>C++ code to execute at start time, before the scheduler setup</td>
<td>No</td>
<td>8-23</td>
</tr>
<tr>
<td>state</td>
<td>Define a state or parameter</td>
<td>No</td>
<td>8-14</td>
</tr>
<tr>
<td>symbol</td>
<td>Associate symbol to a parameter for better documentation</td>
<td>No</td>
<td>8-14</td>
</tr>
<tr>
<td>version</td>
<td>Version number and date</td>
<td>No</td>
<td>8-8</td>
</tr>
<tr>
<td>wrapup*</td>
<td>C++ code to invoke at the end of a run (if no error occurred)</td>
<td>No</td>
<td>8-24</td>
</tr>
<tr>
<td>vendor</td>
<td>Name of company that authors component. All shipped with Agilent Ptolemy are marked Agilent EEsof</td>
<td>No</td>
<td>8-27</td>
</tr>
</tbody>
</table>

* 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 **Table 8-2**, together with subitems of other directives.

**Table 8-2. Directive Subitem**

<table>
<thead>
<tr>
<th>Items</th>
<th>Subitems and Descriptions</th>
<th>Required</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>inmulti, input</td>
<td>name (Name of port or group of ports) type (Data type of input and output particles) descriptor (Summary of function of the input) numtokens (Number of tokens consumed by the port; useful only for dataflow domains)</td>
<td>Yes</td>
<td>8-21</td>
</tr>
<tr>
<td>method, virtual method, inline method, pure virtual method, inline virtual method</td>
<td>name (Name of the method) access (Private, protected, or public) arglist (Arguments to the method) type (Return type of the method) code (C++ code defining the method)</td>
<td>Yes</td>
<td>8-26</td>
</tr>
<tr>
<td>outmulti, output</td>
<td>name (Name of port or group of ports) type (Data type of output particles) descriptor (Summary of functions of output) numtokens (Number of tokens produced by port; useful only for dataflow designs)</td>
<td>Yes</td>
<td>8-21</td>
</tr>
<tr>
<td>state</td>
<td>name (Name of the state variable) type (Data type of the state variable) default (Default initial value; always a string) descriptor (Summary of function of state) attributes (State attributes for simulator) units (Type of dimensional units associated with state) enumlist (list of enumeration options) enumlabels (list of alternate names for enumeration options) extensions (list of file extensions for a filename state - default is txt)</td>
<td>Yes</td>
<td>8-14</td>
</tr>
</tbody>
</table>
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 SDFFIR, but the derivedfrom statement allows you to say either

```
derivedfrom {FIR}
```

or

```
derivedfrom {SDFFIR}
```

The derivedfrom statement may also be written derivedFrom or derived. Note that it is not possible to derive stars across domains.
Writing Component Models

**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 \}\n```

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_SCHEMADISPLAY  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 in the ADS online help under Manuals > Signal Processing components. 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.

```plaintext
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.

```plaintext
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 \{"\text{polar}(3.2, 1.13)\"\}.

An example of the default field for each type is shown in Table 8-3.

- 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).

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>default {3}</td>
</tr>
<tr>
<td>float</td>
<td>default {3}</td>
</tr>
<tr>
<td>fix</td>
<td>default {1.25}</td>
</tr>
<tr>
<td>complex</td>
<td>default {&quot;(1.25, 2.5)&quot;}</td>
</tr>
<tr>
<td>string</td>
<td>default {&quot;string value&quot;}</td>
</tr>
<tr>
<td>precision</td>
<td>default {&quot;24/32&quot;} or equivalently default {&quot;8.24&quot;}</td>
</tr>
<tr>
<td>enum</td>
<td>default {&quot;value 1&quot;}</td>
</tr>
<tr>
<td>filename</td>
<td>default {&quot;/user/abc/xyz/lnm.txt&quot;}</td>
</tr>
<tr>
<td>intarray</td>
<td>default {1, 2, 4, 7, 9}</td>
</tr>
<tr>
<td>floatarray</td>
<td>default {1.25, 3.50, 6.75}</td>
</tr>
</tbody>
</table>
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 Table 8-4.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>complexarray</td>
<td>default (&quot;(1.25, 2.5) (2.4, -2.3) (-1.2, -2.2)&quot;)</td>
</tr>
<tr>
<td>stringarray</td>
<td>default (&quot;Button1&quot; &quot;Button 2&quot;)</td>
</tr>
</tbody>
</table>

Table 8-3. Default Fields (continued)
• The enumlist field is required when the state type is enum. The enumlist field is a comma separated list of strings.

```plaintext
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 underscore (_). For example, "value 1" should be referenced as "value_1".

---

Table 8-4. Unit Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREQUENCY_UNIT</td>
<td>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.</td>
</tr>
<tr>
<td>TIME_UNIT</td>
<td>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.</td>
</tr>
<tr>
<td>ANGLE_UNIT</td>
<td>Results in use of degree angle units at the schematic level. The state in the code will receive a value in terms of degrees.</td>
</tr>
<tr>
<td>POWER_UNIT</td>
<td>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.</td>
</tr>
<tr>
<td>DISTANCE_UNIT</td>
<td>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.</td>
</tr>
<tr>
<td>LENGTH_UNIT</td>
<td>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.</td>
</tr>
<tr>
<td>RESISTANCE_UNIT</td>
<td>Results in use of resistance scale factors at the schematic level (Ohms, KOhms, MOhms, etc.). The state in the code will receive a value in terms of Ohms.</td>
</tr>
<tr>
<td>CAPACITANCE_UNIT</td>
<td>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.</td>
</tr>
<tr>
<td>INDUCTANCE_UNIT</td>
<td>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.</td>
</tr>
</tbody>
</table>
• 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.
Writing Component Models

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 { ARFCON_1_10, ARFCON_2_20, ARFCON_3_30 }
    enumlabels { 10 MHz, 20 MHz, 30 MHz }
    default { ARFCON_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” on page 8-36 and “Array States” on page 8-38.

**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:

```plaintext
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, HPfix, 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 Chapter 9, Data Types for Model Builders. 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:

\[
\text{type } \{= \text{name} \}
\]

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:

```c++
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:

```c++
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” on page 8-31. 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.
Writing Component Models

**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" on page 8-31.

**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" on page 8-31. 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 Agilent 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:

```c
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 Agilent 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 Agilent Ptolemy Star

In general, the task of an Agilent 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 Agilent 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.
Writing Component Models

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 \texttt{setup()} and \texttt{go()} methods. They can be created using the \texttt{Error::warn()} methods. These methods have the following prototypes:

\begin{verbatim}
   Error::warn(const char *, const char * = 0, const char * = 0);
   Error::warn(const NamedObj&, const char *, const char * = 0, const char * = 0);
\end{verbatim}

Where possible, the second method should be used so that the name of the \texttt{NamedObj} can be displayed along with the message. The named object can be \texttt{*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 \texttt{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 \texttt{setup()} method, state values should be checked for any value range error. If an error exists in the state value, the \texttt{State::rangeError()} method should be used:

\begin{verbatim}
   <state_name>.rangeError(const char *);
\end{verbatim}

where the \texttt{const char *} is a string that defines the required state range.

Examples:

\begin{verbatim}
   FCarrier.rangeError(">= 0.0");
   Top.rangeError("> Bottom");
\end{verbatim}

If an error other than this state range error occurs during the initialization process, the \texttt{Error::initialization()} methods should be used. These methods have the following prototypes:

\begin{verbatim}
   Error::initialization(const char *, const char * = 0, const char * = 0);
   Error::initialization(const NamedObj&, const char *, const char * = 0, const char * = 0);
\end{verbatim}

Where possible, the second method should be used so that the name of the \texttt{NamedObj} can be displayed along with the message. The named object can be \texttt{*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:

```cpp
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 chapter 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,

```cpp
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:
Writing Component Models

```cpp
...%0 = currentSample;
```

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:

```cpp
...%1 = pastSample;
%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 any type 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:

```cpp
...%0 = currentSample;
double value = double(currentSample);
```

or more concisely,

```cpp
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:

```cpp
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:

```cpp
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, HPFix, 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 Chapter 9, Data Types for Model Builders.

**SDF (TSDF) PortHole Parameters**

In the preceding example, where in%1 was referenced, some special action is required to tell Agilent 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:

```cpp
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:

```cpp
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:

```cpp
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:

```c++
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 portnamenumberPorts(), 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, FixParticle, and HPFixParticle, which hold data of type int, double (not float), Complex, Fix, and HPFix, 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, Fix, and HPFix. Each particle also has a virtual print method, so a star that writes particles to a file can accept any type.

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 Agilent 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:

```cpp
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:

```cpp
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 SDF Chop star, in which `use_past_inputs` is an integer state,

```cpp
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” on page 9-2 in Chapter 9, Data Types for Model Builders.
A state may be updated by ordinary assignment in C++, as in the following lines:

```cpp
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,

```plaintext
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:

```plaintext
default {"<foo"}
```

where the file `foo` must be located in the current project data subdirectory. If not in the subdirectory, then the filename must include the full directory path as a prefix. For instance:

```plaintext
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:

```plaintext
default {"<foo 2.0"}
default {"0.5 <foo <bar"}
```

A repeat notation is also supported for ArrayState objects: the two value strings

```plaintext
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:

```cpp
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.

```cpp
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:

  ```cpp
  Complex InTSDFPort::getIQData(int n)
  ```

  where \( n \) is the current value of the input stream. For example, if the Timed input port is named \( \text{in} \), then

  ```cpp
  \text{in.getIQData}(0)
  ```

  returns a complex number, which is the current I and Q members of the Timed particle.

  Similarly, the methods

  ```cpp
  int InTSDFPort::getFlavor(int)
  ```

  and

  ```cpp
  double TSDFPortHole::getCarrierFrequency(int)
  ```

  return the Flavor and Fc associated with incoming Timed particle.
Writing Component Models

• Sending data

As described in the preceding section “Reading Inputs and Writing Outputs” on page 8-31, 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.

```plaintext
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 Fc, a TSDF Star method should be used in the star setup as follows:

```plaintext
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

```plaintext
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 TSDF Star method should be used in the star setup, as follows:

```plaintext
TSDFPortHole::setTimeStep(double tstep)
```
• 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}  
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);
}
Writing Component Models

// this method is 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}
  }
}
The next example is the Gain star, which multiplies its input by a constant and outputs the result:

```c

defstar {
    name {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.

```c++

defstar {
    name {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 << ((l*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 Agilent 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.
Writing Component Models

• In writing Agilent 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:

```cpp
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:

```cpp
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

```cpp
puts(savestring(s));
```

should instead be written as

```cpp
const char* newstring = savestring(s);
puts(newstring);
delete [] newstring;
```
Several features in Agilent 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.

Agilent Ptolemy pl File Template

The following is a pl file template:

```plaintext
HPtolemy 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 HPtolemy
  // 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 $}
  // 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 // Optional: delete if not used
  derivedfrom {base_star_name} // Prefix to name of derived star
  copyright {
  Copyright (c) Agilent Technologies 2000
  All rights reserved.
  }
  explanation {my_model_explanation}
  // Use as many lines as needed to describe the star, its 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
    default {my_state_default_value}
    // Example int:                 1
  }
}
```

8-50 Writing Timed Components
// 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 complex array: "(1.25, 2.5) (2.4, -2.3) (-1.2, -2.2)"
// Example string array: "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)
    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.
Writing Component Models

// 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
    // See programmers manual for the use of each port_type
    name {port_name}        // When Domain == SDF:
    type {port_type}        //Options: int, float, fix, complex, message, int_matrix_env,
                              // float_matrix_env, complex_matrix_env, fix_matrix_env,
                              // anytype
    // When Domain == TSD:
    // 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)
}

deructor {
// 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++ 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 Agilent 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 “SDFSimpleBERSink” on page 8-60 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 hindude 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:

```c
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
Writing Component Models

timer/counter reaches \texttt{stop\_value} the task is automatically removed from the TaskList. If you want to remove the task from the TaskList before \texttt{stop\_value} is reached, you can do so by calling the \texttt{stopControl} function. This is used in the BER sinks (see example SimpleBERSink at the end of this section) where \texttt{stopControl} is called when some condition is satisfied. Although in this case you might want to ignore \texttt{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 \texttt{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 \texttt{stop\_value}, use a very large value, e.g., $1.0 \times 10^{20}$. When calling this \texttt{initialize} function, \texttt{start\_value} must be greater than or equal to 0, \texttt{stop\_value} must be greater than or equal to \texttt{start\_value}, and \texttt{step\_value} must be greater than 0. If these conditions are not satisfied the simulation will abort.

2. When the second \texttt{initialize} function is called, the \texttt{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 \texttt{collectData} is called, it increments the timer/counter by \texttt{step\_value} and returns 1 (TRUE) if the timer/counter (before being incremented) was greater than or equal to \texttt{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 \texttt{SinkControl} object is initialized using the first \texttt{initialize} function and \texttt{start\_value}, \texttt{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 \texttt{initialize} function, \texttt{start\_value} must be greater than or equal to 0 and \texttt{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” on page 8-60.

• 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 “SDF MedianSink” on page 8-60. 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” on page 8-29 in this chapter. Refer to “SDF SimpleBERSink” on page 8-60 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:

```cpp
dataP = newSimData(this);
```

• setIndepVar( const char *name, 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 INT (the independent variable will be of integer type; typically used for numeric data) or 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, 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 `INT` (the dependent variable will be of integer type), `FLOAT` (the dependent variable will be of double type), or `COMPLEX` (the dependent variable will be of complex type). The `FIX` data type, as well as all the `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, DataType type, State::Unit unit)` is an overloaded version of the `setDepVar` function that can be used to give the dependent variable the name `baseNamesuffix`. 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 “SDF SimpleNumericSink” on page 8-60, which can handle `INT`, `FLOAT`, and `COMPLEX` data.
Writing Component Models

Examples of Sink Models
This section gives links to five examples of sink models:

SDFSimpleNumericSink
TSDFSimpleTimedSink
SDFSimpleBERSink
SDFMeanVarianceSink
SDF MedianSink

Click on each link 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:

- SDFSimpleNumericSink.pl
- TSDFSimpleTimedSink.pl
- SDFSimpleBERSink.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 MedianSink 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.
Chapter 9: 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 Agilent 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 Agilent Ptolemy kernel. There are four numeric scalar types—complex, fixed-point, double precision floating-point, and integer—described in the section “Scalar Numeric Types” on page 9-1. 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.

Agilent Ptolemy supports user-defined types—using the class Message, described in the section “Defining New Data Types” on page 9-20. 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” on page 9-27.

With Agilent 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” on page 9-42. There is also the timed signal type with two forms, Baseband and Complex Envelope, described in the section, “Timed Particle Signal Type” on page 9-43.

Scalar Numeric Types

There are four scalar numeric data types defined in the Agilent 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” on page 8-31 in Chapter 8, Writing Component Models. 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 Agilent 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 Agilent 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:

```cpp
double real() const
double imag() const
```

**Non-Member Functions and Operators**

The following one- and two-argument operators return a new complex number:

```cpp
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.

```cpp
Complex conj (const Complex& x)
```

Return the complex conjugate of the number.

```cpp
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:

```cpp
double abs(const Complex& x)
```
Data Types for Model Builders

Return the absolute value, defined to be the square root of the norm.

\[
\text{double arg(const Complex& x)}
\]

Return the value \(\arctan(x.imag()/x.real())\).

\[
\text{double norm(const Complex& x)}
\]

Return the value \(x.real()^2 + x.imag()^2\).

\[
\text{double real(const Complex& x)}
\]

Return the real part of the complex number.

\[
\text{double imag(const Complex& x)}
\]

Return the imaginary part of the complex number.

Comparison Operators:

\[
\text{int operator != (const Complex& x, const Complex& y)}
\]

\[
\text{int operator == (const Complex& x, const Complex& y)}
\]

The Fixed-Point Data Type

The fixed-point data type is implemented in Agilent 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:

\[
x.y \text{ or } y/n
\]

where

\[
x = \text{number of integer bits (including sign bit) to the left of the decimal point}
\]

\[
y = \text{number of fractional bits to the right of the decimal point}
\]

\[
n = \text{total number of bits (bitwidth)}
\]
Fixed-point operations include consideration of overflow type (wrapped, saturate, saturate-to-zero) and quantization type (round or truncate).

Agilent 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.

<table>
<thead>
<tr>
<th>Attributes for Fix Data Type</th>
<th>UCB Fixed-Point Stars</th>
<th>Agilent Fixed-Point Stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArithType, default</td>
<td>2's complement</td>
<td>2's complement</td>
</tr>
<tr>
<td>ArithType, options.</td>
<td>2's complement</td>
<td>Synthesizable DSP—2's complement, unsigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-synthesizable DSP—2's complement</td>
</tr>
<tr>
<td>Precision, max bit width</td>
<td>32</td>
<td>256</td>
</tr>
<tr>
<td>Overflow handler, default</td>
<td>saturate</td>
<td>Synthesizable DSP—wrapped</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-synthesizable DSP—saturate</td>
</tr>
<tr>
<td>Overflow handler, options</td>
<td>wrapped, saturate, saturate-to-zero</td>
<td>Synthesizable DSP—wrapped, saturate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-synthesizable DSP—wrapped, saturate, saturate-to-zero</td>
</tr>
<tr>
<td>RoundFix, default</td>
<td>truncate</td>
<td>truncate</td>
</tr>
<tr>
<td>RoundFix, options.</td>
<td>round, truncate</td>
<td>round, truncate</td>
</tr>
<tr>
<td>Generates Verilog or VHDL</td>
<td>No</td>
<td>Synthesizable DSP—Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-synthesizable DSP—No</td>
</tr>
</tbody>
</table>
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:

```c++
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” on page 9-9).
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.

Fixed-Point Inputs and Outputs

Fix types are available in Agilent 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 Agilent Ptolemy Fix type for use with synthesizable DSP components. The user must explicitly cast an int or double particle to a synthesizable fix particle (HPFix) using a Signal Converter (IntToHPFix or FloatToHPFix).

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:

```plaintext
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:

```plaintext
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:

```plaintext
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}
    }
}
```
defstate {
    name {UseArrivingPrecision}
    type {int}
    default {"YES"}
    desc {
    Flag indicating whether or not 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 {
    SDFFix::begin();

    if (! int(UseArrivingPrecision)) {
        fixIn = Fix((const char *) InputPrecision);
        if(fixIn.invalid())
Data Types for Model Builders

```cpp
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");

// 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 SDFFix
  // if you defined your own, you should call SDFFix::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.
Data Types for Model Builders

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:
Data Types for Model Builders

```cpp
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:

```cpp
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**

```cpp
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:

```cpp
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

```cpp
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:

```cpp
void set_overflow(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:

```cpp
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:

```cpp
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 Agilent 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 Agilent 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. 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:

#ifndef VECTOR_H_INCLUDED
#define VECTOR_H_INCLUDED

// @(#) $Source:
//cvs/wlv/src/hptolemy/hptolemy/src/examples/message/Vector.h,v $ $Revision:
100.7 $ $Date: 1998/08/05 18:37:22 $

// Not required, unless compiling under g++. These directives will
// not effect other compilers.
#ifdef __GNUG__
#pragma interface
#endif

#include "Message.h"
#include "hpvectorDll.h"

// Data type of the new message type. This variable name must be in
all capital letters. The ptlang preprocessor will convert the port
type field into all capital letters.*/
DllImport extern const DataType VECTOR;

// A vector of doubles. Example of a user defined data type in HP
// Ptolemy.
Data Types for Model Builders

```cpp
class Vector:public Message {
public:
    // Default Constructor
    Vector();

    // Copy Constructor
    Vector(const Vector&);

    // Destructor
    ~Vector();

    // Return the data type of the Message
    DataType type() const {
        return 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 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;
Data Types for Model Builders

```c
// The size of the array
int sz;
```

```c
#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:

2. Copy the files SDFConst_V.pl, SDFVectToFloat.pl, SDFVectToMx.pl, Vector.h, Vector.cc, hpvectorDll.h and make-defs in the directory
   ```
   $HPEESOF_DIR/modelbuilder/examples/hptolemy/message
   ```
   into the network directory of your new project.
3. Compile the simulator following the directions in Chapter 7, Building Signal Processing Models.

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 message example for their exact types. The base class conversion functions assert a run-time error, and the default print function returns a `StringList` reading:
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 (see the programming example for UnPackInt below). 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 Agilent 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” on page 9-20.

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++ doubles.)

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 Agilent 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 Agilent Ptolemy's PtMatrix class with Fortran library calls. The limits of interfacing Agilent Ptolemy's PtMatrix class with other software is discussed in the section “Public Functions and Operators for the PtMatrix Class” on page 9-28.

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. Agilent 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 Agilent 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 numRow, 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.
Data Types for Model Builders

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 numRow, 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 numRow, 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.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.
Subtract the scalar value from each entry in the matrix.
Data Types for Model Builders

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. Note: this is an elementwise operation and is not equivalent to A = A * B.

XXXMatrix& operator *=(xxx value)

Example: A *=value.
Multiply each matrix entry by the scalar value.

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.
Divide each matrix entry by the scalar value. This value must be non-zero.

XXXMatrix& operator identity()

Example: A.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 = ~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.transpose().
   This is the same as the ~ operator but called by a function name instead of as an operator.

XXXMatrix inverse()
   Example: B = A.inverse().
   This is the same as the ! operator but called by a function name instead of as an operator.

ComplexMatrix conjugate()
   Example: ComplexMatrix B = A.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: ComplexMatrix B = A.hermitian().
   Return a new matrix that is the Hermitian Transpose (conjugate transpose) of the source. This function is defined for the ComplexMatrix class only.
Data Types for Model Builders

**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.
Data Types for Model Builders

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 Agilent 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

```cpp
#include "{HPtolemyMatrix.h}"
```

needs to be in the Star definition. Otherwise, the declaration

```cpp
#include "{HPtolemyMatrix.h}"
```

is sufficient.

To declare an input porthole that accepts matrices, the following syntax is used:

```cpp
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 Agilent 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:

```cpp
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" on page 9-25.) 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” on page -25.

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” on page 9-21).

Matrix Outputs

To put a matrix into an output porthole, the syntax is:

```c
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” on page 9-25), 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:

```c
defstar {
    name {Add_M}
    domain {SDF}
    desc {Output is the sum of all the floating-point input matrices.}
    location {Numeric Matrix}
    inmulti { input }
    name {input}
```
type {FLOAT_MATRIX}
}
output {
  name {output}
  type {FLOAT_MATRIX}
}
ccinclude {"HPtolemyMatrix.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 Agilent 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:

```python
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

Agilent 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, \(\text{flavor}\) 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:

\[
V(t) = \Re\left\{v(t) e^{j2\pi F_c t}\right\}
\]

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 \(F_c\) 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 \(F_c\) 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 $F_c$ 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 ptag 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.
Data Types for Model Builders

```csharp
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 tstep;
tstep = out1.getTimeStep();
TStep = tstep;
}
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);
      }
    }
  }
Chapter 10: Porting UC Berkeley Ptolemy Models

A major design goal of Agilent 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.

Agilent Ptolemy currently supports most SDF UCB Ptolemy stars (component models). The porting process for one of these stars is relatively straightforward. In this chapter 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 project outlined in the beginning of Chapter 7, Building Signal Processing Models. 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 Chapter 7, Building Signal Processing Models.
Tcl/Tk Porting Issues

There are two differences in Tcl/Tk use in Agilent Ptolemy vs. UCB Ptolemy. The first is that Agilent 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 Agilent 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. Agilent Ptolemy has its own methods for parameter sweeping and optimization, as described in earlier chapters.

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 Agilent Ptolemy Message documentation in "Defining New Data Types" on page 9-20 in Chapter 9, Data Types for Model Builders. The Message class infrastructure has been greatly simplified and enhanced in Agilent 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 Agilent 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" on page 9-20 in Chapter 9, Data Types for Model Builders.

An example vector Message class implementation complete with type conversion stars is provided in Agilent Ptolemy. See the example in the section in "Defining New Data Types" on page 9-20 in Chapter 9, Data Types for Model Builders.

**Porting Procedures**

- Using the Vector message class example, modify your message class to conform to the new Agilent Ptolemy Message class. Refer to "Defining a New Message Class" on page 9-21 in Chapter 9, Data Types for Model Builders, for more information. This should be a relatively straightforward 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 Agilent 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.
Index

A
A_CONSTANT attribute, 8-16
A_NONCONSTANT attribute, 8-16, 8-36
A_NONSETTABLE attribute, 8-16
A_SETTABLE attribute, 8-16, 8-36
accessMessage method
   MessageParticle class, 9-26
Add (SDF block), 8-35
AddFix (SDF block), 9-7
AEL expressions
   and User-Compiled Analog Models, 1-49
   changes from Series IV, 1-49
   replacement of Series IV data items, 1-47
   aggressive reclamation, 9-26
Agilent Ptolemy
   porting UC Berkeley models into, 10-1
   algorithm examples, 2-18
   Analog Model Development Kit
      file management, 1-50
      porting from Libra Senior, 1-47
   ANSI-C code, 1-2
   anytype portholes, 8-21
   ArrayState class, 8-38
   ArrivingPrecision parameter, 9-10
   attributes
      for Agilent Ptolemy model development, 8-9, 8-10, 8-14, 8-16, 8-36

B
bad format parameters
   Fix class, 9-6
behavioral models, 6-1
binary point, 9-6
BJ Ts
   SDDs, 5-36
Boltzmann constant
   and User-Compiled Analog Models, 1-30
   defining macros for, 1-30

C
capacitors
   implementing with SDDs, 5-47
   SDDs, 5-25
   using in transient models, 4-2
 carriers
   FDDs, 6-19
   ccinclude hpeesoflang directive, 8-25
   characteristics
      of User-Compiled Analog Models, 1-28
   circuit elements
      creating interface definitions and declarations, 1-30
   Circuit Envelope simulator
      of FDDs, 6-3
      sources, 6-5
   circuits
      nodal analysis, 5-45
   clearAttributes method, 8-40
   clock enables
      adding to FDDs, 6-17, 6-27
clean method
   Message class, 9-24, 9-26
coaxial cable example, 2-19
code hpeesoflang directive, 8-25
coding
   linear elements, 2-6
   compile-time scheduling, 8-23, 8-24
   compiling
      models, 1-22
Complex class, 8-36, 8-37, 9-2-9-4
   - operator, 9-3
   != operator, 9-4
   * operator, 9-3
   *= operator, 9-2
   + operator, 9-3
   += operator, 9-2
   / operator, 9-3
   /= operator, 9-2
   -= operator, 9-2
   == operator, 9-4
   abs() function, 9-3
   arg() function, 9-4
   basic operators, 9-2
   conj() function, 9-3
   constructors, 9-2
   cos() function, 9-3
   exp() function, 9-3
   imag() function, 9-3, 9-4
   log() function, 9-3

Index-1
norm() function, 9-4
pow() function, 9-3
real() function, 9-3, 9-4
sin() function, 9-3
sqrt() function, 9-3
Complex data type, 9-2, 9-4
complex state, 8-15
complex type
portholes, 8-21
states, 8-15
COMPLEX_MATRIX_ENV, 9-39
complex_matrix_env type
portholes, 8-21
complexarray type
states, 8-15
ComplexArrayState class, 8-36
ComplexMatrix, See Matrix class
ComplexParticle class, 8-36
ComplexState class, 8-36, 8-37
components
FDD examples, 6-22
SDD examples, 5-17, 5-47
conscalls hpeesoflang directive, 8-23
constitutive relationships, 5-4
FDDs, 6-8
SDDs, 5-4
constructor hpeesoflang directive, 8-22
constructors, 8-23
continuity
FDDs, 6-9
SDDs, 5-7
controlling currents
SDDs, 5-10, 5-15, 5-34
convergence
FDDs, 6-9
copy constructor
Message class, 9-24
creating
linear elements, 2-1
nonlinear elements, 3-1
current equations
FDDs, 6-8
custom nonlinear devices, 5-1
data items
See also Series IV data items
referencing in User-Compiled Analog Models, 1-43
data types, 8-21
user-defined, 9-20
default parameter values, 8-15
default value for states, 8-15
delay
for matrix arcs, 9-40
in dataflow, 9-40
in FDDs, 6-19
descriptor, 8-16
destructor hpeesoflang directive, 8-23
diodes
SDDs, 5-28
user-defined model, 3-4
divide by zero
Fix class, 9-6
dummy message, 9-26
E
element files
user-defined and User-Compiled Analog Models, 1-46
element parameters
and User-Compiled Analog Models, 1-31
keyword strings, 1-31
types, 1-31
element responses
nonlinear, 1-34
transient, 1-33, 1-35
elements
linear and User-Compiled Analog Models, 1-36
nonlinear and responses, 1-34
user-defined in User-Compiled Analog Models, 1-46
using built-ins, 1-45
using in User-Compiled Analog Models, 1-45
empty method
Envelope class, 9-25
enumerated states, 8-15
Envelope class, 9-20, 9-25
envelope waveforms, 6-4
equation based
nonlinear devices, 5-1, 6-1
equations
adding to FDDs, 6-24
constitutive relationships, 5-4
deriving Y-parameters, 2-4
FDDs, 6-8
for SDDs, 5-3
nodal analysis, 5-45
port, 5-10
error messages
of SDDs, 5-50
error/warning messages
displaying in User-Compiled Analog Models, 1-44
examples
FDDs, 6-28
IQ modulator, 6-29
mixer, 6-31
sample and hold, 6-34
SDDs, 5-17
examples of sink models, 8-60
explicit representations
of SDD equations, 5-4

F

FDDs

about, 6-1
applications, 6-1
clock enables, 6-17, 6-27
compared to SDDs, 6-1
components, 6-22
constitutive relationships, 6-9
continuity, 6-9
delaying carriers, 6-19
equations, 6-8
examples, 6-22, 6-28
freq parameter, 6-10, 6-25
functions, 6-6, 6-18
harmonics, 6-11
impedance, 6-21
indexing, 6-6
IQ modulator, 6-29
miscellaneous functions, 6-20
mixer, 6-31
overview, 6-2
port variables, 6-6, 6-18
procedure for adding, 6-23
sample and hold, 6-34
signal models, 6-2
simulation compatibility, 6-21
simulations, 6-21

trigger applications, 6-1
trigger events, 6-16, 6-18
triggers, 6-26

file management

in Analog Model Development Kit, 1-50
filename states, 8-15
Fix class, 9-4-9-19
- operator, 9-17
* operator, 9-17
* operator, 9-17
+= operator, 9-16
+ operator, 9-17
-= operator, 9-16
_/ operator, 9-17
/= operator, 9-17
-= operator, 9-16
clear_errors(), 9-16
compare(), 9-16
complement(), 9-18
constructors, 9-13
conversion operators, 9-18
dbz(), 9-16
intb(), 9-14
invalid(), 9-16
is_zero(), 9-15
len(), 9-14
max(), 9-15
maximum length, 9-6
min(), 9-15
overflow(), 9-14
ovf_occurred, 9-16
precision(), 9-14
roundMode(), 9-15
set_overflow, 9-15
set_rounding, 9-15
setToZero(), 9-15
signBit(), 9-15
uninitialized, 9-9
value(), 9-15

fix type

portholes, 8-21
states, 8-15
FIX_MATRIX_ENV, 9-39
fix_matrix_env type
portholes, 8-21

fixed-point, 9-4

array parameters, 9-7
inputs and outputs, 9-7
Index-4

parameters, 9-7
precision, 8-15
setting precision, 8-15
states, 9-7
fixed-point data type, 9-4
FixMatrix, See Matrix class
float type
portholes, 8-21
states, 8-15
FLOAT_MATRIX_ENV, 9-39
float_matrix_env type
portholes, 8-21
floatarray type
states, 8-15
FloatArrayState class, 8-36, 8-38
FloatMatrix, See Matrix class
FloatParticle class, 8-36
FloatState class, 8-36
Fork (SDF block), 8-34
freq
adding to FDDs, 6-25
FDDs, 6-10
frequency-domain defined devices. See FDDs
functions
harm variable, 6-11
and User-Compiled Analog models, 1-32
for FDDs, 6-7, 6-16, 6-17, 6-20
weighting
SDDs, 5-8

G
Gain (SDF block), 8-45
getMessage method
MessageParticle class, 9-26
go hpeesoflang directive, 8-24
Gummel-Poon BJ T
SDDs, 5-36

H
harmonic balance simulations
of FDDs, 6-2
harmonics
FDDs, 6-11
indexing using _harm, 6-11
header hpeesoflang directive, 8-26
heterogeneous message interface, 9-20
hinclude hpeesoflang directive, 8-25
hpeesoflang See model development, 8-24
I
ideal amplifiers
SDDs, 5-20
ideal mixers
SDDs, 5-23
impedance
defining for FDDs, 6-21
implicit representation
of SDD equations, 5-5
indexing
FDDs, 6-6, 6-10
freq, 6-10
using _harm, 6-11
inductors
SDDs, 5-32
using in transient models, 4-2
initialized Fix objects, 9-9
initializing states from files, 8-38
inmulti hpeesoflang directive, 8-21, 8-34
input hpeesoflang directive, 8-21, 8-31
InSDFPort class, 8-34
int type
portholes, 8-21
states, 8-15
INT_MATRIX_ENV, 9-39
int_matrix_env type
portholes, 8-21
intarray type
states, 8-15
IntArrayState class, 8-36
interface
creating declarations and definitions, 1-30
IntMatrix, See Matrix class
IntParticle class, 8-36
IntState class, 8-36

L
Libra Senior
porting to Analog Model Development Kit, 1-47
library browser
adding User-Compiled Analog Models, 1-26
linear elements
and User-Compiled Analog Models, 1-36

Index-4
Index-5

using built-ins in User-Compiled Analog Models, 1-45
linear models creating, 2-1
linear pi-section attenuator example, 2-9
linking
User-Compiled Analog Models, 1-14

M
macros
in Boltzmann constant, 1-30
in User-Defined Analog Models, 1-30
in userdefs.h file, 1-30
Matrix class, 9-27-9-41
- operator, 9-36
- operator, unary negation operator, 9-34
! operator, inverse operator, 9-35
!= operator, 9-32
* operator, 9-36
*=( operator, 9-34
+ operator, 9-36
+= operator, 9-33
/ operator, 9-34
-= operator, 9-33
= operator, assignment operator, 9-33
== operator, 9-32
^ operator, 9-35
~ operator, transpose operator, 9-34
cloned() function, 9-37
ComplexMatrix, 9-28
conjugate() function for ComplexMatrix, 9-35
constructors, 9-29
conversion operators, 9-32
dataType() function, 9-38
display() function, 9-29
FixMatrix, 9-28
FixMatrix, special constructors, 9-31
FloatMatrix, 9-28
hermitian() function for ComplexMatrix, 9-35
including Matrix.h into a Star, 9-39
identity() function, 9-34
IntMatrix, 9-28
inverse() function, 9-35
isA() function, 9-38
multiply() function, 9-37
outputting to a PortHole, 9-40

print() function, 9-37
star input and output, 9-39
transpose() function, 9-35
writing Stars that use the Matrix class, 9-38
Matrix.h include file, 9-39
memory leaks, preventing, 8-47
Message class, 9-20
message data type, 8-21
message type
ports, 8-21
MessageParticle class, 8-36, 9-20, 9-26
messages
displaying error/warning messages in
User-Compiled Analog Models, 1-44
method hpeesoflang directive, 8-26
mixers
SDDs, 5-23
model development
signal processing
command-line method, 7-1
creating a simple model library, 7-4
debugging, 7-14
platform-specific issues, 7-16
prerequisites to model development, 7-3
src directory and make-defs in detail, 7-11
write a model, 7-13

Model Development Kit
signal processing, See model development, signal processing
models
diode example, 3-4
FDDs, 6-2
nonlinear parts, 3-2
transient user defined, 4-1
modified nodal analysis
SDDs, 5-45
MultiOutSDFPort class, 8-34
multiple portholes, 8-34
multiport hole class, 8-34
myData method
Envelope class, 9-25

N
narrow-band models, 6-22
nodal analysis, 5-45
noise
analysis in linear models, 2-7
behavior in linear models, 2-7
characteristics
adding to a linear model, 2-14
parameters in linear models, 2-7
thermal, 2-14
thermal in linear models, 2-7
nonlinear elements
  equation-based, 5-1
  responses, 1-34
  SDD capacitors, 5-25
  SDD inductors, 5-32
  SDD resistors, 5-18
nonlinear models, 6-1
diode example, 3-4
  making, 3-1
  parts of, 3-2
n-ports
  FDDs, 6-6
numberPorts method, 8-35
numtokens hpeesoflang directive, 8-21
O
operator, referencing an entry, 9-29
outmulti hpeesoflang directive, 8-21, 8-34
output hpeesoflang directive, 8-21, 8-33
OutSDFPort class, 8-34
outSDFport class, 8-31
overflow
  Fix class, 9-6
P
parameters
  complex, 8-15
  Particle class, 9-20
  particle class, 8-31, 8-36
  particle types, 8-36
  pl file template, 8-50
  polymorphism, 8-47
  porthole class, 8-31
  portholes
    modifying in derived classes, 8-40
porting
  matrix stars, 10-4
  message class, 10-3
  Tcl/Tk issues, 10-1, 10-2
  UC Berkeley models, 10-1
ports
  constitutive relationships, 5-4
  current equations, 6-8
  equations
    SDDs, 5-10
    hiding from the user, 8-40
    SDDs, 5-10
    variables, 5-3
    FDDs, 6-6
    voltage equations, 6-8
    precision
      parameter in Agilent Ptolemy, 9-6
    precision type
      states, 8-15
  print method, 8-36
    Message class, 9-25
  Printer (SDF block), 8-46
  private hpeesoflang directive, 8-25
  protected hpeesoflang directive, 8-25
  public hpeesoflang directive, 8-25
Q
quantization
  Fix class, 9-6
R
Ramp (SDF block), 8-44
reference count, 9-20, 9-25, 9-38
resistors
  using in transient models, 4-2
  rounding
    Fix class, 9-6
S
sample and hold, 6-17
saturation
  Fix class, 9-6
SDDs
  about, 5-1
  adding to a schematic, 5-13
  and mixers, 5-23
  capacitor example, 5-25
  compared to FDDs, 6-1
  continuity, 5-7
  controlling currents, 5-10, 5-15, 5-34
  diode example, 5-28
  equations, 5-10
  error messages, 5-50
examples, 5-17
Gummel-Poon BJT example, 5-36
ideal amplifier example, 5-20
implementing capacitors, 5-47
modified nodal analysis, 5-45
nodal analysis, 5-45
nonlinear inductors, 5-32
nonlinear resistor, 5-18
port variables, 5-3
weighting function, 5-8
weighting functions, 5-16
SDFFix class, 9-12
Series IV
  AEL expressions changes, 1-49
Series IV data items
    replacement by AEL expressions, 1-47
setAttribute method, 8-40
setInitValue method, 8-40
setSDFParams method, 8-21, 8-33
setup hpeesoflang directive, 8-23
sign bit, 9-6
simulations
  FDDs, 6-21
  of FDDs, 6-2
sink models
  examples, 8-60
  writing, 8-54
sources
  FDDs, 6-5
S-parameter equations, 2-1
state hpeesoflang directive, 8-36
states
  hiding from the user, 8-40
  modifying in derived classes, 8-40
string states, 8-15
stringarray states, 8-15
StringArrayState class, 8-36
StringState class, 8-36
substrate components
  User-Compiled Analog Models, 1-48
substrates
  use model See substrate components
symbolically-defined device. See SDDs
  symbols
  creating or modifying, 1-20
T
templates
for user-compiled models, 1-23
timed components, writing, 8-41
transient elements
  defining, 4-5
  responses, 1-33
  transient models
    user defined, 4-1
  transient responses, 1-35
transmission lines
  example, 2-15
  using in transient models, 4-3
trigger events
  FDDs, 6-16
  functions, 6-18
  triggers
    adding to FDDs, 6-26
  truncation
    Fix class, 9-6
tuned models, 6-22
two's complement, 9-6
type conversion, 8-36
  Message class, 9-24
U
underflow
  Fix class, 9-6
uninitialized Fix object, 9-9
user defined nonlinear devices, 5-1
User-Compiled Analog Models
  adding to library browser, 1-26
  and Boltmann constant, 1-30
  changes to AEL expressions, 1-49
  characteristics, 1-28
  creating new models, 1-16
  deleting, 1-14
  element parameter types, 1-31
  element parameters, 1-31
  functions, 1-32
  keyword strings, 1-31
  linking, 1-14
  macros, 1-30
  opening an existing model, 1-13
  porting from Libra Senior, 1-47
  referencing data items, 1-43
  substrate components
  writing and compiling, 1-22
user-compiled signal processing models,
See model development, signal processing
user-defined elements
booting in User-Compiled Analog Models, 1-46
files and User-Compiled Analog Models, 1-46
user-defined messages, 9-21
userdefs.h file
function declarations, 1-30
interface data structure type definitions, 1-30
macros, 1-30
symbols, 1-30

V
variables
_harm, 6-11
_for SDD ports, 5-3
vector message, 9-21
voltage equations
FDDs, 6-8

W
weighting functions
SDDs, 5-8, 5-16
wrapup hpeesoflang directive, 8-24
writing sink models, 8-54
writing data to dataset, 8-58

Y
Y-parameters
equations, 2-4