



Agilent Technologies

Advanced Design System 2002
Agilent Ptolemy Simulation

February 2002

Notice

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

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

Warranty

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

Restricted Rights Legend

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

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

Copyright © 2002, Agilent Technologies. All Rights Reserved.

Acknowledgments

Portions of the documentation

Copyright © 1990-1996 The Regents of the University of California. All rights reserved.

In no event shall the University of California be liable to any party for direct, indirect, special, incidental, or consequential damages arising out of the use of this software and its documentation, even if the University of California has been advised of the possibility of such damage. The University of California specifically disclaims any warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The software provided hereunder is on an “as is” basis and the University of California has no obligation to provide maintenance, support, updates, enhancements, or modifications.

Contents

1 Overview of Agilent Ptolemy	
Agilent Ptolemy Major Benefits.....	1-1
Agilent Ptolemy Major Features	1-1
Agilent Ptolemy and UC Berkeley Ptolemy.....	1-1
Timed Synchronous Dataflow Simulator	1-2
Terminology	1-2
What's in this Manual.....	1-3
2 Integrator Example	
Copying and Opening Example Project.....	2-1
Copy the Project.....	2-1
Open the Project	2-2
Selecting and Placing Components.....	2-3
Add a Source.....	2-3
Add an Output Display.....	2-4
Modify Component Parameters.....	2-5
Connect Components with Wires	2-5
Add a Controller	2-6
Starting Simulation	2-7
Simulate and Display Data Directly	2-7
Simulate and Save Data.....	2-8
3 Data, Controllers, Sinks, and Components	
Representation of Data Types	3-1
Single Versus Multiple Arrowheads	3-1
Automatic or Manual Data Type Conversion.....	3-2
What Happens During Conversion?	3-3
Controllers	3-5
Controls Tab	3-6
Options Tab	3-7
Resistors Tab.....	3-9
Debug Tab	3-10
Display Tab.....	3-11
Sources and Sinks Control the Simulation	3-11
Multiple DF Controllers on the Schematic Are Not Allowed	3-13
Agilent Ptolemy Components	3-15
4 Understanding Parameters	
Value Types	4-1
Parameter Editing	4-7
Parameter Expressions	4-8

Complex-Valued Parameters	4-9
Parameters for Fixed-Point Components.....	4-9
String Parameters.....	4-17
Filename Parameters	4-17
Array Parameters.....	4-17
Reading Array Parameter Values From Files.....	4-18
Parameters With Optimization and Swept Attributes	4-19
5 Using Data Types	
Representation of Data Types	5-1
Stem Thickness	5-2
Single and Multiple Arrowheads.....	5-2
Data Types Defined	5-3
Numeric Scalar Data	5-3
Numeric Matrix Data.....	5-4
Timed Data.....	5-4
Conversion of Data Types.....	5-5
What Happens During Conversion?	5-5
Numeric Scalar and Matrix Conversions	5-5
Timed Data Conversions	5-6
Rules and Exceptions.....	5-8
Automatic or Manual Data Type Conversion	5-8
6 Understanding File Formats	
Real Array Data	6-1
Complex Array Data	6-1
String Array Data.....	6-1
Float Matrix Data	6-1
Fixed-Point Matrix Data	6-2
Integer Matrix Data.....	6-2
Complex Matrix Data	6-2
SPW (.ascsig and .sig) File Formats	6-3
Time-Domain Waveform Data (.tim) File, MDIF ASCII Format.....	6-4
Time-Domain Waveform Data (.bintim) File in Binary Format.....	6-5
Agilent Standard Data Format (.dat) Files	6-6
7 Performing Parameter Sweeps	
Introduction.....	7-1
Sweeping Parameters Using the ParamSweep Component	7-1
Procedure.....	7-2
Sweeping Parameters Using the VAR Component	7-4
Procedure.....	7-5
Using a Sweep Plan	7-6

Procedure	7-6
Sweeping Various Parameter Types	7-9
Sweeping a Complex Waveform Component	7-10
Performing Multidimensional Sweeps	7-12
8 Using Nominal Optimization	
Optimizing Various Parameter Types	8-1
Optimizing Input and Output Bit Width	8-2
Setting Up the VAR Component	8-4
Setting Up the ConstInt, Second Sink, and Second Goal Components	8-5
Completing the Optimization	8-6
9 Theory of Operation	
Introduction	9-1
Synchronous Dataflow (SDF)	9-2
Basic Dataflow Terminology	9-2
Balancing Production and Consumption of Tokens	9-3
How Schedulers Work in Agilent Ptolemy	9-4
Iterations in SDF	9-6
Inconsistency	9-6
Deadlocks	9-7
Deadlock Resolution	9-8
Timed Synchronous Dataflow (TSDF)	9-8
Time Step Resolution	9-8
Carrier Frequency Resolution	9-9
Input/Output Resistance	9-9
References	9-10
10 Introduction to MATLAB Cosimulation	
Setting Up MATLAB	10-1
Simulating with MATLAB	10-2
Writing Functions for the MATLAB Models	10-3
Examples	10-4
11 Cosimulation with Analog/RF Systems	
Setting Up the Analog/RF Circuit Schematic	11-2
Setting Up the Signal Processing Schematic	11-3
Circuit Simulation Controllers	11-4
Numeric-to-Timed Converters	11-4
Clustering of Circuit Subnetworks	11-4
Connected Circuit Subnetworks	11-5
Connected Resistors	11-5
Feedback Loops	11-5
Named Connections and Measurements in Circuit Designs	11-6

Circuit Envelope Specific Rules.....	11-7
Transient Simulation Specific Rules	11-7
Nested Simulation Approach	11-7
Signal Processing Model of the Circuit Network	11-8
Circuit Model of the Signal Processing Network	11-8
Interface Issues	11-8
Time Step	11-9
Delays in Feedback Loops	11-10
Time Converters.....	11-10
Carrier Frequency.....	11-10
EnvOutSelector and EnvOutShort Components	11-11
Troubleshooting Common Problems.....	11-12
Cosimulation Example.....	11-13
Copying and Opening the Project.....	11-13
Rectifier Schematic	11-14

12 Using Interactive Controls and Displays

List of Interactive Control and Display Components.....	12-2
TkSlider and TkPlot Components Example.....	12-3
Placing Multiple TkSlider Components.....	12-5
TkText and TkShowValues	12-6
TkXYPlot Component	12-7
TkBarGraph.....	12-9
LMS Adaptive Filter Components.....	12-9
TkButtons	12-11
TkBreakPt.....	12-12
TkMeter	12-12
TkShowBooleans.....	12-12
Baseband Equivalent Channel	12-13
Invoke Tcl Script	12-14
TkEye, TkConstellation, TkHistogram, TklQrms, and TkPower.....	12-14
Additional Resources on Tcl/Tk.....	12-14
Books	12-14
World Wide Web.....	12-15

13 Building Signal Processing Models

Advanced Model Building Functions	13-1
Prerequisites to Model Development.....	13-2
Creating a Simple Model Library	13-3
Set the HPEESOF_DIR and PATH Environment Variables	13-3
Set Up the Area to Build Models	13-3
Write a Model	13-4
Edit the make-defs.....	13-5

Build the Shared Library	13-5
Build the AEL, Default Bitmaps, and Default Symbols	13-6
Simulate Your Model	13-7
Sharing Your Stars	13-8
The src Directory and make-defs in More Detail	13-8
Variables	13-8
Dependencies	13-9
Debugging Your Model	13-10
Running Simulations from the Command Line	13-11
Debugging Under Windows	13-11
Debugging Under UNIX	13-12
Platform-Specific Issues	13-12
HP-UX 10.x	13-12
HP-UX 11.x	13-12
Windows	13-13
AIX	13-13
14 Writing Component Models	
Using the Agilent Ptolemy Preprocessor Language	14-1
Rectangular Pulse Star Example	14-1
Items Defining a defstar	14-3
Writing C++ Code for Stars	14-21
The Structure of an Agilent Ptolemy Star	14-21
Messaging Guidelines for Star .pl Files	14-21
Reading Inputs and Writing Outputs	14-24
Modifying PortHoles and States in Derived Classes	14-32
Writing Timed Components	14-33
Programming Examples	14-36
Preventing Memory Leaks in C++ Code	14-39
Agilent Ptolemy pl File Template	14-41
Writing Sink Models	14-45
Understanding Tasks	14-46
Sink Coding Methodology	14-46
Useful Notes/Hints	14-48
Writing Data to a Dataset	14-49
Examples of Sink Models	14-51
15 Data Types for Model Builders	
Scalar Numeric Types	15-1
The Complex Data Type	15-2
The Fixed-Point Data Type	15-4
Defining New Data Types	15-19
Defining a New Message Class	15-20

Use of the Envelope Class	15-24
Use of the MessageParticle Class	15-25
The Matrix Data Types.....	15-25
Design Philosophy.....	15-26
The PtMatrix Class.....	15-27
Public Functions and Operators for the PtMatrix Class.....	15-27
Writing Stars and Programs Using the PtMatrix Class.....	15-36
Writing Stars That Manipulate Any Particle Type	15-40
Timed Particle Signal Type.....	15-40
Time Step Resolution.....	15-43
Carrier Frequency Resolution.....	15-44
16 Porting UC Berkeley Ptolemy Models	
Tcl/Tk Porting Issues.....	16-1
Porting Procedures.....	16-2
Message Class.....	16-2
Porting Procedures.....	16-2
Matrix Stars.....	16-3
17 Glossary	
Terminology.....	17-1
Index	

Chapter 1: Overview of Agilent Ptolemy

Part of Advanced Design System, Agilent Ptolemy software gives you the simulation tools you need to evaluate and design modern communication systems products. Today's designs call for implementing DSP algorithms in an increasing number of portions in the total communications system path, from baseband processing, to adaptive equalizers and phase-locked loops in the RF chain. Agilent Ptolemy software gives you the most complete tool set available to do your job—including cosimulating with the Advanced Design System's RF and analog simulators from the same schematic.

Agilent Ptolemy Major Benefits

The Agilent Ptolemy simulator allows you to:

- Find the best design topology using state-of-the-art technology with over 500 behavioral DSP and communication systems models
- Cosimulate with RF and analog simulators
- Integrate your intellectual property from previous designs
- Reduce the time-to-market for your products

Agilent Ptolemy Major Features

- Timed Synchronous Dataflow simulation
- Cosimulation capability with RF and analog simulators
- Easy-to-use interface for adding and sharing your own custom models
- Interface to test instruments
- Data Display with post-processing capability
- Integration with ADS DSP Synthesis

Agilent Ptolemy and UC Berkeley Ptolemy

The Ptolemy signal processing simulator has its roots at the University of California at Berkeley. UC Berkeley Ptolemy is a third-generation software environment that began in January of 1990. It is an outgrowth of two previous generations of design

environments, Blossim and Gabriel, that were aimed at digital signal processing. Both environments use dataflow semantics with block-diagram syntax for the description of algorithms.

Agilent EEsof has built on the UC Berkeley Ptolemy code in developing Agilent Ptolemy software. We have added a large number of behavioral, time-domain models, which are critical to communication systems designers. These include antenna and propagation models. For DSP designers, we have upgraded fixed point analysis to be scalable up to 256 bits. Agilent Ptolemy software runs under Advanced Design System’s intuitive user interface, which includes post-processing capability, cosimulation with analog/RF simulators, links to test instruments, online help, and a host of other features.

In Ptolemy, different specialized design environments are called *domains*. Agilent Ptolemy has modified the proven Synchronous Dataflow domain to include timed components. This domain is called the Timed Synchronous Dataflow domain.

Timed Synchronous Dataflow Simulator

The Timed Synchronous Dataflow domain captures years of Agilent EEsof expertise in system-level analog/RF simulation, while adding the benefits of dataflow technology. This domain enables fast RF simulation, integration with signal processing simulation, and cosimulation with Agilent EEsof circuit simulators. For more information on the Timed Synchronous Dataflow simulator and the Synchronous Dataflow domain, refer to [Chapter 9, Theory of Operation](#).

Terminology

Throughout most of the Agilent Ptolemy documentation, we use the Advanced Design System terminology, which is standard EDA terminology. However, UC Berkeley Ptolemy has its own terminology and for users familiar with this terminology, or those who are writing their own models, the following table compares the terms. Only in the chapters on building signal processing models, and in the theory of operation chapter, is the UC Berkeley Ptolemy terminology used.

Table 1-1. Terminology Comparison

Agilent Ptolemy	UC Berkeley Ptolemy
Component	Star
Network (or circuit)	Galaxy

Table 1-1. Terminology Comparison (continued)

Agilent Ptolemy	UC Berkeley Ptolemy
Top-level System	Universe
Controller	Target
Wire	Arc
Data (or signals)	Particles (or tokens)

What's in this Manual

The goal of this manual is to teach you how to use the Agilent Ptolemy simulator, how to cosimulate with Analog/RF Systems designs, and how to build your own signal processing models for use with Agilent Ptolemy.

In addition, the *Signal Processing Components* online help and the *User's Guide* are available to help answer questions you may have.

This manual contains:

- Chapter 1—Overview of Agilent Ptolemy
- Chapter 2—Integrator Example (new user tutorial)
- Chapter 3—Data, Controllers, Sinks, and Components
- Chapter 4—Understanding Parameters
- Chapter 5—Using Data Types
- Chapter 6—Understanding File Formats
- Chapter 7—Performing Parameter Sweeps
- Chapter 8—Using Nominal Optimization
- Chapter 9—Theory of Operation
- Chapter 10—Introduction to MATLAB Cosimulation
- Chapter 11—Cosimulation with Analog/RF Designs
- Chapter 12—Using Interactive Controls and Displays
- Chapter 13—Building Signal Processing Models
- Chapter 14—Writing Component Models
- Chapter 15—Data Types For Model Builders

- Chapter 16—Porting UC Berkeley Ptolemy Models
- Chapter 17—Glossary
- Index

Chapter 2: Integrator Example

This chapter is designed for the new user. If you know how to Advanced Design System for Analog/RF Systems design, you can skim this chapter noting the differences to Agilent Ptolemy, such as the use of sinks, Data Flow controller, and Interactive Controls and Displays components.

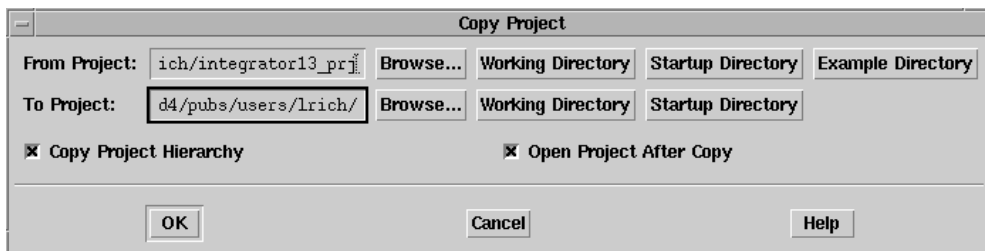
To learn how to use Agilent Ptolemy, let's load a simple integrator example. We will add a source, an output display item, and a controller, then simulate and view the results.

Copying and Opening Example Project

First, we will copy an example project.

Copy the Project

1. From the Main window, choose **File > Copy Project**. A dialog box appears.



Note On UNIX platforms, you do not generally have permission to work in the Advanced Design System Example directories. You must copy the example project to a directory for which you have write permission. On PC platforms, while you can work in the Example directories if you want, it's better to copy the examples to another directory.

2. In the From Project field, click the **Examples Directory** button, and then the **Browse** button. The File Browse dialog box appears with the Example directories listed.
3. Scroll to the bottom of the Directories list and select the **/Tutorial** directory.

4. Select **integrator_prj** from the list of files in the Files field.
5. In the To Project field, click the **Startup Directory** or **Working Directory** button (depending on where you want to copy the project to) or choose the **Browse** button to select another directory.
6. Choose **Copy Project Hierarchy**. This ensures that all the appropriate directories and files will be copied.
7. Click **OK** to copy the project and close the dialog box.

Open the Project

1. From the Main window, choose **File > Open Project**. When the Open Project dialog box appears, select *<the directory you copied the example to>* in the Directories field, then double-click **integrator_prj** in the Files field. The project will appear in the File Browser field of the Main window.
2. Under the **integrator_prj** project directory, click the Networks subdirectory to open the various schematics within this project. These will all have the extension *.dsn*.
3. Double-click **integrator1.dsn**. In the Design Information field at the right, one item appears.
4. Double-click **integrator1** (Schematic) to open the design. The schematic appears:

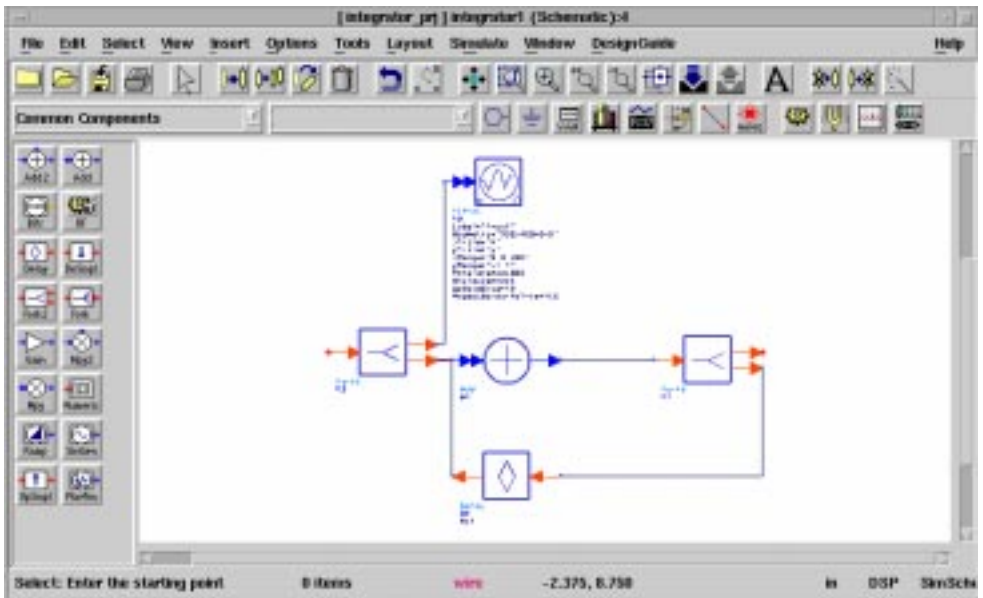


Figure 2-1. Integrator Design

Selecting and Placing Components

We will add a sine wave source, an output display item, and a controller to the integrator schematic. There are two ways to choose library components:

- From a Palette List The far left side of the window contains a palette list and icons for each item in the palette. You choose items by first selecting a palette and then clicking on the icons in the palette.
- From a Library You can also select components by choosing *Insert > Component > Component Library*. A dialog box appears that displays components in each component library. You select items from the list.

Add a Source

1. We will use the Palette List method first (left side of the window). Since the component we want is in the default library, called Common Components, simply click the **SinGen** icon (near bottom of list). Crosshairs and a ghost image of the component appear as you move the pointer over the design window.

2. Move the crosshairs to the upper left part of the schematic (to the left of the Fork2 component), then click *once*. A schematic representation of the source component is placed in the design window. Beneath the schematic is block representing the component's name and editable parameters. We will accept the default values.
3. If you are finished placing components, click the **End Command** arrow on the toolbar, or press **Escape**. The crosshairs disappear. If you are continuing to place components, as in this example, you can skip this step.

Note If you continue to click without deselecting, you will place a new component with each click.

Add an Output Display

We will continue by adding the output display item. But this time we will use the Library method of selecting components.

1. Choose **Insert > Component > Component Library**. A dialog box appears that displays components in each component library. From the Libraries list box, select **Interactive Controls and Displays** (resize the dialog box to show long names).
2. From the right side, select **TkPlot**. Crosshairs and a ghost image of the component appear as you move the pointer over the design window. (Another TkPlot item is already in the schematic to display the input signal.)
3. Move the crosshairs to the upper right part of the schematic (to the right of the Fork2 component), then click *once*. A schematic representation of the TkPlot display component is placed in the design window. Beneath the schematic is an options block representing the component's name and editable parameters.
4. As explained before, click the **End Command** arrow on the toolbar, or press **Escape**. The crosshairs disappear.
5. Close the Component Library dialog box by choosing **OK**.

Modify Component Parameters

We will modify two parameters for this item. There are several ways to edit parameters, including:

- Double-click the top of the component.
- Choose **Edit > Component > Edit Component Parameters**.
- Click the Edit Component Parameters button on the toolbar.
- Type the parameter value directly on the schematic page. The text changes color. Then edit the value and press **Return** at the right of the new value. Pressing Return also takes you through subsequent parameters.

We will use the dialog box method.

1. Double-click the **TkPlot** item. A dialog box appears.
2. Select the xRange parameter (left side). On the right side, backspace over the 100 and type **400**.
3. Select the yRange parameter (left side). On the right side, backspace over the -1.5 1.5 and type **0 32**.
4. Similarly, select the Persistence parameter and change the value from 100 to **300**.
5. Type **Output** in the Label field (top of list) so we can keep track of the input and output plots.
6. Choose **OK**.

Connect Components with Wires

1. Choose **Insert > Wire** or click the **Insert Wire** button on the toolbar (bottom row). Connect a wire from the port on the **SineGen** source to the input port on the **Fork2** component. When a port is successfully connected, its color changes from red to blue.
2. Connect a wire from the top port of the **Fork2** component to the port on the TkPlot display component.

Note Wires must connect ports in pairs, and you must place at least two components before you can add a wire. You cannot add a wire to a component port first, and then

add a second component to that wire.

Add a Controller

Controllers are used to specify the type of simulator you want to use and simulation parameters.

1. From the Palette List under Common Components, select the **Data Flow Controller** icon (right, near top). Crosshairs and a ghost image of the component appear as you move the pointer over the design window.
2. Move the crosshairs to the lower left part of the schematic, then click once.
3. A schematic representation of the controller component is placed in the design window. Controllers are not connected or wired to other components. We will accept the default values.
4. Click the deselect arrow, or press **Escape**. The crosshairs disappear.

There are several types of controllers, the one we have chosen is called the Data Flow controller, which is used to run mixed numeric and timed signal processing simulations.

At this point, your example should look similar to [Figure 2-2](#).

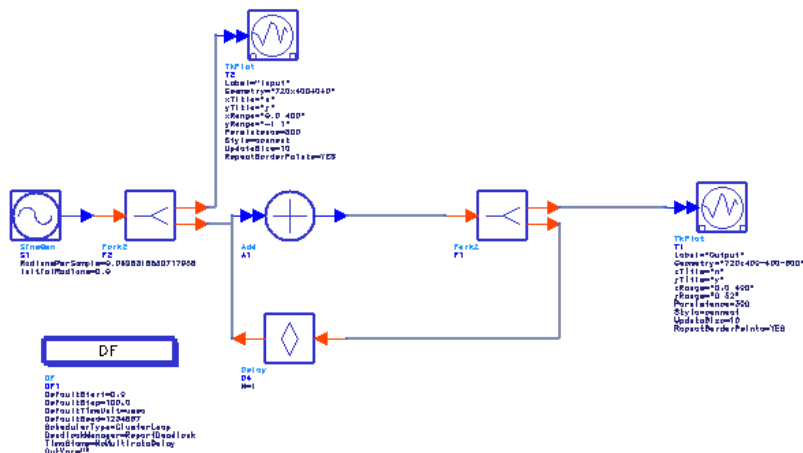


Figure 2-2. Integrator Design After Source, Display, and Controller Items Added.

If you have had difficulty building the design, you can select the completed schematic from your directory you copied the example project to earlier. Select *integrator1_complete.dsn*.

Starting Simulation

Now that we have a completed schematic, we're ready to start a simulation. The Advanced Design System provides flexibility in this task. In our example, we have placed an interactive display item called TkPlot. This item quickly displays the results of your simulation. Later we will substitute a "Sink" item in the schematic that will save the simulation results to a file. We will then use the Data Display to review our results.

Simulate and Display Data Directly

1. Choose **Simulate > Simulate**. The simulation begins. A status window appears, which gives you information on your simulation or reports errors.
2. Two TkPlot windows appear showing you an animated display of both the sine wave input and the simulation results of the output, as shown below.

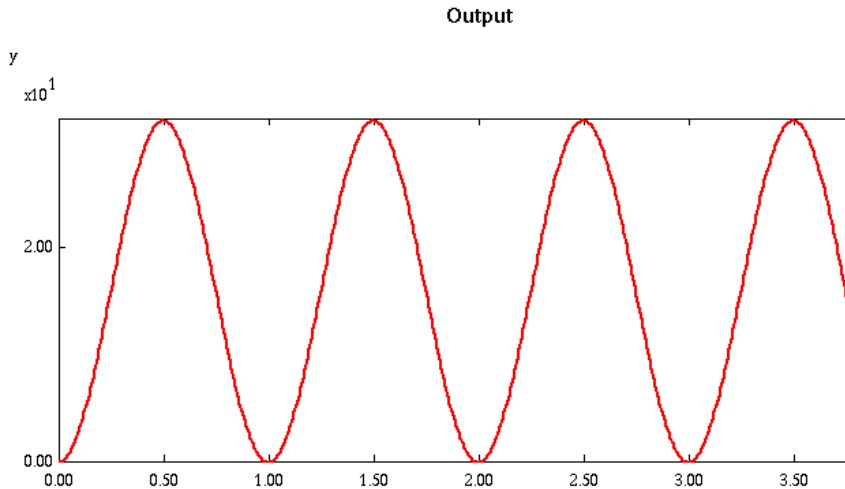


Figure 2-3. Integrator Output Simulation Results

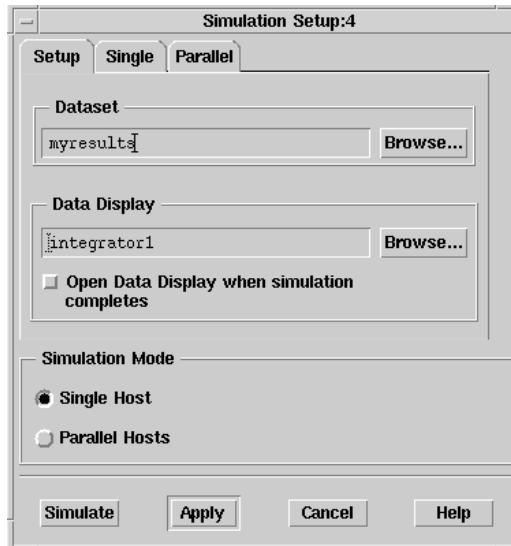
In this simple example, the sine wave source has been changed to a cosine wave (offset by 90 degrees) by the integrator.

The simulation must be stopped manually. Choose **Quit** from the small dialog box labeled Agilent Ptolemy when you are done reviewing the animated plots.

Simulate and Save Data

Now we will use the alternate approach where we substitute a “Sink” component in the schematic and save the data.

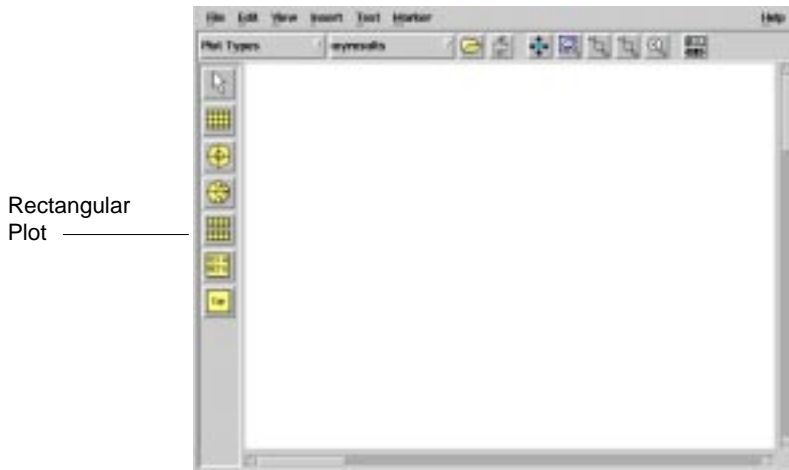
1. Click the *output TkPlot* item in your schematic to select it.
2. Press the **Delete** key or choose the Delete (trash can) icon from the toolbar.
3. From the **Common Components** Palette List, select the **Numeric** icon (NumericSink). Crosshairs appear.
4. Place the **NumericSink** where the TkPlot item was originally.
5. Double-click the **NumericSink** to edit the sink’s parameters.
6. Accept the Start default of **DefaultNumericStart**.
7. Select Stop and change the value to **200**. Here we show that a sink can override the Data Flow controller’s Stop value. Typically, a sink can be left at its default and you can control simulation from the Data Flow controller.
8. Choose **OK**.
9. Choose **Simulate > Simulation Setup**. The Simulation Setup dialog box appears. This step is used when you want to explicitly name a dataset.



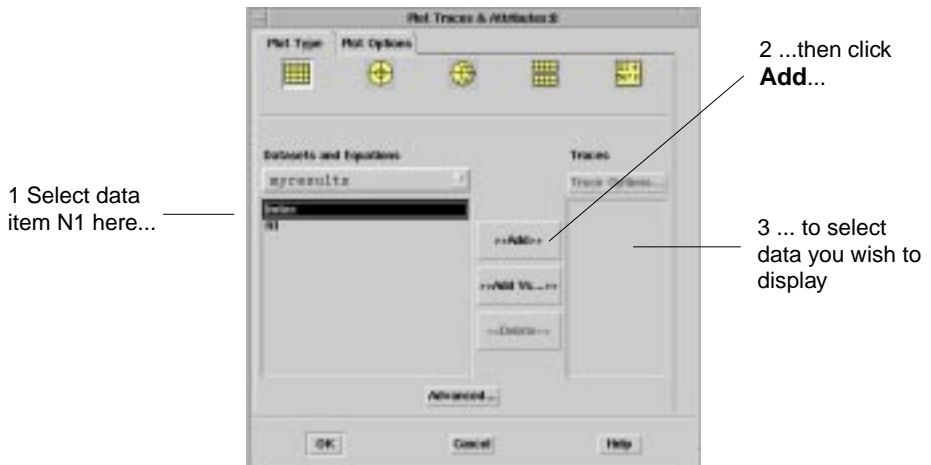
10. In the Dataset field, type **myresults**. This becomes the filename of your simulation results. Leave the other defaults as they are.
11. Choose the **Simulate** button (at bottom). The simulation begins. A status window appears, which gives you information on your simulation or reports errors.

This time, your data is saved to disk where it can be used to display results in a variety of formats, or be used in post-processing procedures. In addition, the input TkPlot displays an animated plot for the input. Click **Quit** in the Agilent Ptolemy controller to dismiss this display.

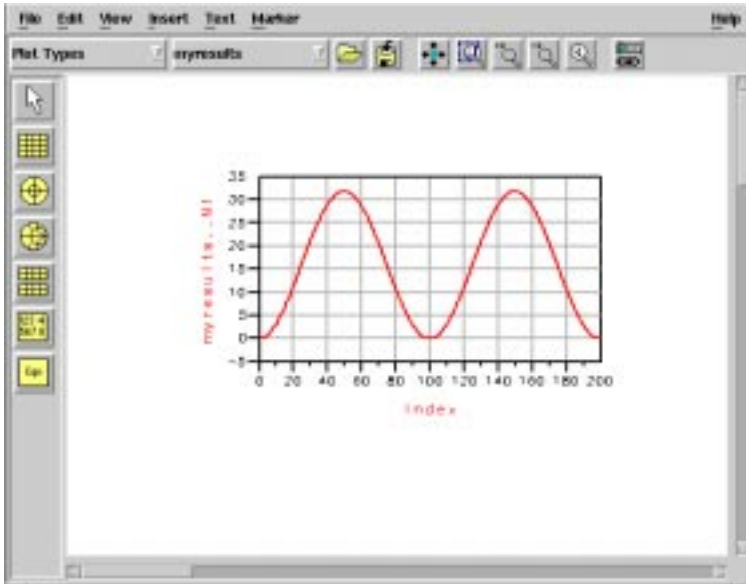
1. Choose **Window > New Data Display**. The Data Display window opens.
2. From the drop-down list next to Plot Types, select **myresults**. This list is called the Default Dataset list.



3. Click **Rectangular Plot** in the Plot Types palette list, at the left of the window. A ghost rectangular frame appears.
4. Click once to place the frame in the Data Display window. A dialog box appears; follow the instructions in the graphic below:



5. Choose **OK**. Your data is plotted in the Data Display window, as shown in the next figure.



To resize a plot, use the various zoom buttons in the toolbar, or drag a corner outward. A large variety of graphing, annotation, and post-processing tasks can be done from this window. Giving the data a unique name allows it to be archived as a reference in a suite of simulations.

We have seen two methods for displaying data, both of which start with the placement of an output item in your schematic:

- TkPlot (one of several interactive display items), which does not store data to disk.
- The Data Display window, which takes stored data and displays it in a variety of formats.

Chapter 3: Data, Controllers, Sinks, and Components

Before continuing to use Agilent Ptolemy, let's look at some of the Agilent Ptolemy concepts you may have questions about and introduce the signal processing components that Agilent Ptolemy uses.

Representation of Data Types

Agilent Ptolemy schematics contain component stems with single and double arrowheads as well as stems of different colors and thicknesses. This differs from the Analog/RF Systems schematics in ADS. [Table 3-1](#) describes the data types and their representation in the Advanced Design System. A note on terminology: For some applications, particularly those using timed components, data types can be thought of as signal types. Regardless of the terminology, packets of data are passed from one component to another.

Table 3-1. Data Type Representation

Data Type	Stem Color	Stem Thickness
Scalar Fixed Point	Magenta	Thin
Scalar Floating Point	Blue	Thin
Scalar Integer	Orange	Thin
Scalar Complex	Green	Thin
Scalar Timed	Black	Thin
Matrix Fixed Point	Magenta	Thick
Matrix Floating Point	Blue	Thick
Matrix Integer	Orange	Thick
Matrix Complex	Green	Thick
Any Type	Red	Thin

Single Versus Multiple Arrowheads

[Figure 3-1](#) shows the thicker stem width associated with matrix data (top) compared to the thinner stem width associated with scalar data (bottom).

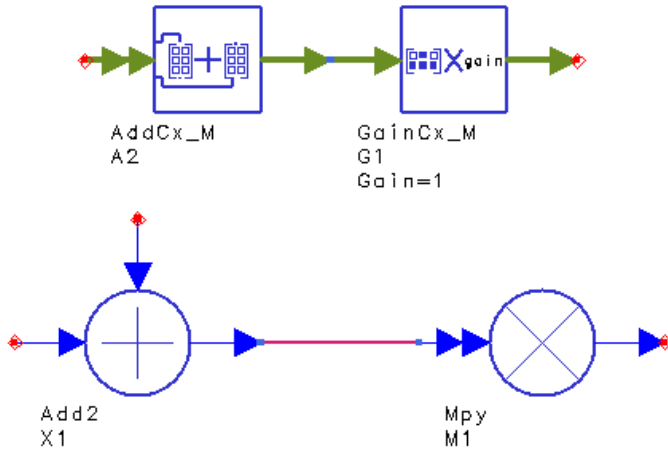


Figure 3-1. Matrix Data (Thick Lines) Versus Scalar Data (Thin Lines)

Components appear on the schematic with single or multiple arrowheads at inputs or outputs.

- Single arrowheads carry only one distinct signal.
- Multiple arrowheads carry more than one distinct signal.

For example, an adder component (Add) has multiple arrowheads on the input pin, but a single output arrowhead.

BusMerge items can be used to connect multiple signals to a component when the signal order must be specified. Similarly, BusSplit items can be used to split signals to multiple outputs.

Automatic or Manual Data Type Conversion

When you connect components of the same data type (color), the data is copied from one component to another. If you connect components represented by different data types, such as complex to floating point, or scalar integer to matrix integer, you need to consider two things about conversion:

Should I place a conversion component in the schematic or let the software automatically do the conversion? What will happen to my data? These questions are answered next.

Although the software will automatically convert dissimilar data types, such as complex to float, we recommend that you place an appropriate converter (from the Signal Converters library) in your schematic. This acts as a visual reminder that a conversion is taking place, and also helps you decode error messages that may arise. Automatic conversion means that an appropriate converter is “spliced in” behind the scenes and is not shown on the schematic.

Automatic conversion is allowed among scalar data types and among matrix data types, but *not* between scalar and matrix data types.

Timed Component Exceptions

For Timed pins, there are two cases when automatic splicing produces an error message:

1. When either a Float to Timed, Fixed to Timed, Integer to Timed, or Complex to Timed converter is placed (or spliced) in the design *and* there is no time step defined (via sources or other timed converters) in the design. You need to define the time step at least once in your design.
2. When a Complex port is connected to a Timed port. *Automatic* conversion from Complex to Timed is not supported. You need to place a Complex to Timed converter between the ports and enter appropriate parameters.

Conversion Between Scalar and Matrix Types

When a scalar pin is directly connected to a matrix pin (or vice versa), without a Pack or Unpack converter, an error message is generated.

In the Numeric Matrix Library, four converters are used to “pack” scalar data into matrix data, such as Pack_M and PackCx_M. Similarly, four converters “unpack” the data (back to scalar), such as UnPk_M and UnPkCx_M. There is no automatic conversion between scalar and matrix data (or vice versa). You must place the converters where needed in your design.

What Happens During Conversion?

Most conversions do what you expect. For example, when converting from lower precision to higher precision data types, such as integer to float, no data is lost; only the format is changed.

When converting from higher precision to lower precision data types, such as float to integer, the outcome is governed by your computer's math rounding rules, with the following exceptions:

Complex Data Conversions

- **Complex to Float**—Agilent Ptolemy computes the magnitude and ignores the phase.
- **Complex to Fixed**—After computing the float magnitude, Agilent Ptolemy converts the float to fixed.
- **Complex to Integer**—After computing the float magnitude, Agilent Ptolemy converts the float to integer.

Timed Data Conversions

The Timed data type represents the time-domain signal in either carrier-modulated (complex) or real-baseband flavors. The Timed data class members are I, Q, F_c, time, plus an Agilent Ptolemy member called Flavor. Flavor specifies whether the Timed data type is in a carrier-modulated or real-baseband format. When the carrier frequency is not specified (undefined) for a Timed port, an error message is generated.

You can convert between Timed and non-Timed ports by placing one of the following converters and supplying the parameters as needed:

- Timed to Complex or Complex to Timed
- Timed to Float or Float to Timed
- Timed to Fixed or Fixed to Timed
- Timed to Integer or Integer to Timed

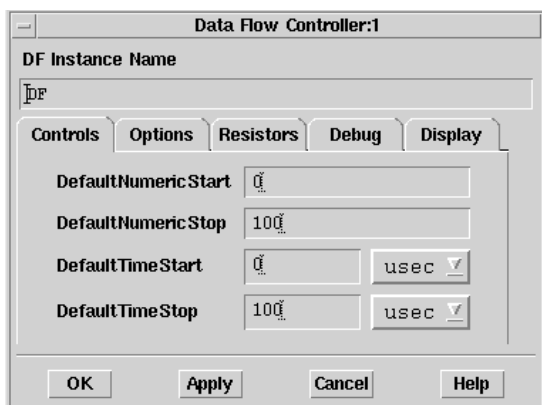
Time-data conversion depends on the flavor of the Timed data and the carrier frequency.

For more detailed information on conversion of data types, refer to [“Conversion of Data Types” on page 5-5](#).

Controllers

Controllers are used to control simulation, are placed unconnected anywhere in the schematic, and are found in the Controllers library or palette. The main Agilent Ptolemy controller is the Data Flow controller, and it is required for all simulations. This controller, together with the source and sink components, provide you the flexibility to control the duration of simulation globally or locally. Other controllers are used to set up parameter sweeps, optimization, or statistical design.

To set or modify the parameters using a dialog box, double-click the component in the schematic, or choose *Edit > Component > Edit Component Parameters*. The Data Flow controller dialog box has five tabs: Controls, Options, Resistors, Debug, and Display, as shown below.



The following information will help you understand the controller's use:

Controls Tab

Agilent Ptolemy sinks have Start and Stop parameters that control when to start and stop data collection. Sinks collect from Start to Stop, inclusively.

In numeric sinks, these numbers are unitless because they represent sample numbers. The first datum that the sink receives is #0, the second is #1, etc. For example, a numeric sink with Start=3 and Stop=4 will skip the first three pieces of data and collect the next two.

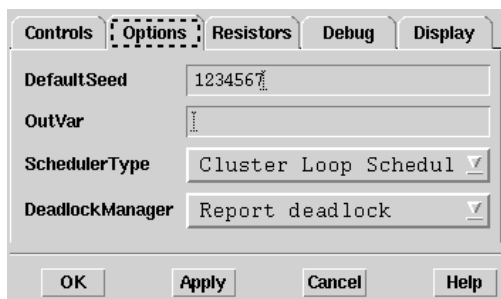
In timed sinks, Start and Stop have timed units because the data has a time base. The amount of data that the sink collects is a function of both the data time base and the sink's Start and Stop parameters. For example, if Start=0 msec, Stop=1 msec, and the data has a time base of 2 usec, the sink will collect 501 pieces of data.

The Controls tab contains global parameters that are the default values for the sink's start and stop parameters. Numeric sinks' start and stop parameters are set to **DefaultNumericStart** and **DefaultNumericStop**. Timed sinks' start and stop parameters are set to **DefaultTimedStart** and **DefaultTimedStop**. The default values for these Data Flow controller values are 0, 100, 0 usec, and 100 usec, respectively.

Sinks can control simulation locally with their own start and stop times, or they can use the appropriate Data Flow parameter to inherit control. By default, all sinks inherit start and stop times from the controller. You can inherit none, one, or both of the start and stop times.

Because these Data Flow parameters function as variables inside the simulation, they can be used inside expressions or overridden in a hierarchal fashion. For example, you could set a numeric sink's parameters to Start=DefaultNumericStart and Stop=DefaultNumericStop*2.

Options Tab



The screenshot shows a dialog box with five tabs: Controls, Options (selected), Resistors, Debug, and Display. The Options tab contains the following fields:

- DefaultSeed**: A text input field containing the value "1234567".
- OutVar**: An empty text input field.
- SchedulerType**: A dropdown menu with "Cluster Loop Schedul" selected.
- DeadlockManager**: A dropdown menu with "Report deadlock" selected.

At the bottom of the dialog are four buttons: OK, Apply, Cancel, and Help.

The options tab has the following parameters:

DefaultSeed Enter an integer to seed the random number generator. The default is 1234567.

The DefaultSeed parameter is used by all random number generators in the simulator, except those components that use their own specific seed parameter. DefaultSeed initializes the random number generation. The same seed value produces the same “random” results, thereby giving you predictable simulation results.

To generate repeatable “random” output from simulation to simulation, use any positive seed value. If you want the output to be truly random, enter the seed value of 0.

OutVar OutVar is a space-separated list of variable names defined using variables and equations (VAR) components. The values of these variables will be sent to the Data Display window. In the case of hierarchical designs, in order to send variables that are at a level other than the top-most level, use the complete path to the variables, which must be period ‘.’ delimited.

Example:

```
OutVar="freq1 freq2 X1.amplitude X2.X4.temp"
```

In this case, there are four variables to be sent to the Data Display: freq1, freq2, amplitude, and temp, each separated with a space. The variable amplitude is contained in subnetwork X1, while the variable temp is contained in subnetwork X4, which in turn is contained in subnetwork X2. These subnetworks are delimited with periods.

Note ADS puts a set of quotes around the OutVar variable. Do not type your own quotes or the double set will cause simulation to fail.

SchedulerType Select a Scheduler Type from the drop-down list. The choices are:

- Cluster Loop Scheduler (default)—Optimized for multirate graphs with feedback cycles.
- Classical Scheduler—Better for uni-rate graphs with cycles.
- Acyclic Loop Scheduler—Better for multirate graphs with no cycles.

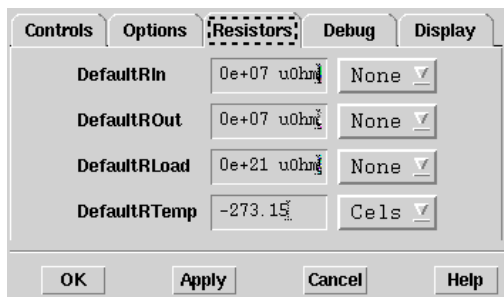
The Scheduler Type option allows you to run the simulation based on one of three options. No matter which scheduler is chosen, the simulation results will be the same. The difference is in the time and memory needed to set up the simulation schedule. It's best to start with the default, and experiment with the others as needed. For more information on these schedulers, refer to [“How Schedulers Work in Agilent Ptolemy” on page 9-4](#).

DeadlockManager The Deadlock Manager allows you to manage deadlocks in designs. A deadlock in a design occurs when a feedback loop does not have a delay in its feedback path, or when a Delay item does not initialize the proper number of signal tokens. A static schedule (required for simulation) can only be derived in a design with no schedule deadlocks.

Select the type of deadlock management from the drop-down list:

- *Report deadlock* (default) indicates the design includes deadlocks.
- *Identify deadlocked loops* allows the user to spot which loops are deadlocked. These loops can be highlighted on the schematic page by double-clicking on the error message or Status window.
- *Resolve deadlock by inserting tokens* will add delays to deadlock loops and allow the simulator to proceed.

Resistors Tab



The image shows a dialog box titled "Resistors" with four tabs: "Controls", "Options", "Resistors", "Debug", and "Display". The "Resistors" tab is active. It contains four rows of controls:

Parameter	Value	Unit
DefaultRIn	0e+07	uOhm
DefaultROut	0e+07	uOhm
DefaultRLoad	0e+21	uOhm
DefaultRTemp	-273.15	Cels

At the bottom of the dialog are four buttons: "OK", "Apply", "Cancel", and "Help".

The Resistors tab controls global parameters related to resistor behavior. As in the Controls tab, these parameters act as variables inside the simulation. Overriding the resistor values in a hierarchal fashion can be especially useful. For example, a large design can have a subcircuit representing a component being tested. By setting the DefaultRTemp inside the Data Flow controller to -273.15, and placing a VAR block with a DefaultRTemp setting inside the subcircuit, you can easily add resistor noise to the subcircuit and nowhere else.

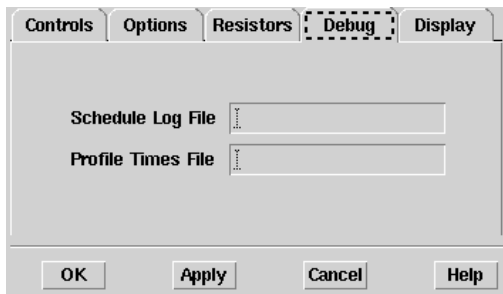
DefaultRIn is the default input impedance of timed components. Its value is 50 ohms.

DefaultROut is the default output impedance of timed components. Its value is also 50 ohms.

DefaultRLoad is the default input impedance of timed sinks and the default impedance of solitary resistors (the R component). Its value is 1.0e15 ohms, representing an infinite load.

DefaultRTemp is the default temperature of resistors. Its value is -273.15 Celsius (0 K), so by default there is no thermal noise.

Debug Tab



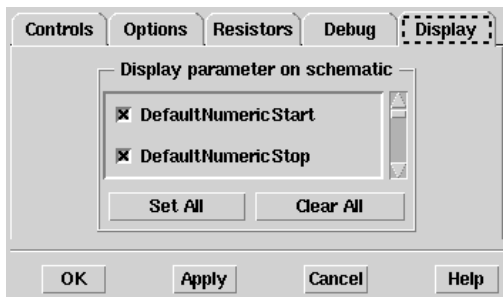
The Debug tab can be used to provide the ability to debug your design and its custom components. There are two fields in this tab:

- **Schedule Log File**
- **Profile Times File**

You can specify a filename in the *Schedule Log File* field. After a simulation is finished, the log file you specified will be generated under the */data* directory of the project. It will log the firing schedule of the components in your design.

You can also specify another filename in the *Profile Times File* field. After a simulation is finished, the file will also be located under the */data* directory of the project. It provides runtime information on the components in your design during simulation. For example, the number of times a component is fired or the average time.

Display Tab



From the display tab, you can choose which parameters will be displayed on the schematic. By default, only some of the parameters will be displayed on the schematic:

- DefaultNumericStart
- DefaultNumericStop
- DefaultTimedStart
- DefaultTimedStop

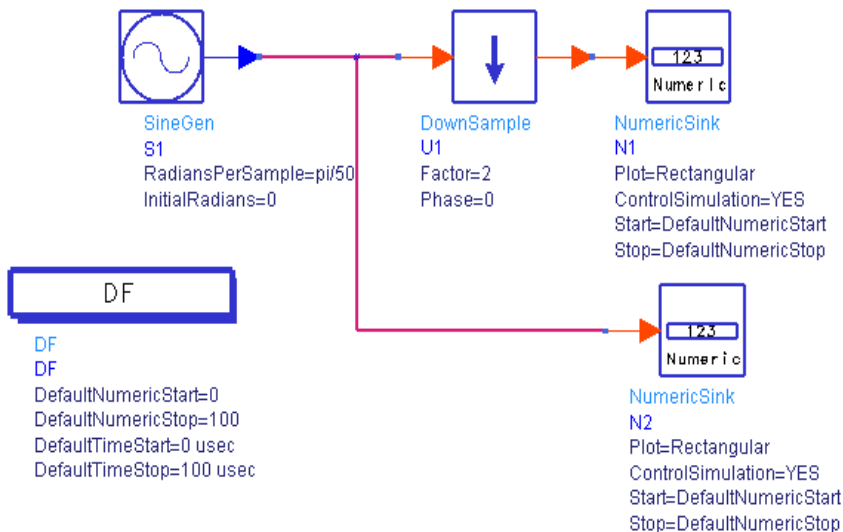
You can add or subtract parameters you want to be displayed on the schematic by scrolling through the list.

Sources and Sinks Control the Simulation

Agilent Ptolemy simulation is controlled by the sources and sinks you place on your schematic. All sinks and many sources have a `ControlSimulation` parameter that is either YES or NO. Controlling sinks and sources keep the simulation running, and non-controlling sinks and sources do not. There must be at least one source or sink that is controlling the simulation.

Sinks

Sinks are components with no outputs. When a sink is controlling the simulation, it will keep the simulation running long enough to satisfy its start and stop times. (One or both of the start and stop times might be inherited by the Data Flow controller.) By default, a sink's `ControlSimulation` parameter is set to YES. When a sink is not controlling the simulation, it will start collecting data at Start, and then collect as much data as the simulation produces. Consider the following example:



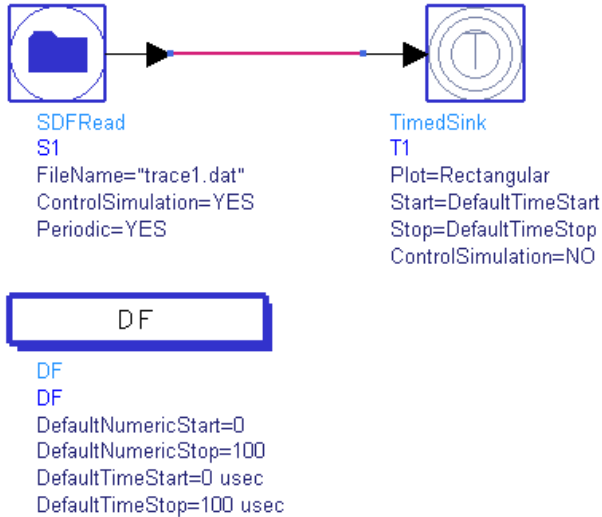
As shown, both sinks will collect 101 data samples (0 to 100 inclusive). They are both controlling sinks so they will obey their start and stop times. Because of the DownSample, sink N2 will receive more data but it will not collect it.

Changing one of the sinks ControlSimulation parameters to NO will cause N2 to collect twice as much data as N1. If N1 is the controller, then it will collect 101 samples, and N2 will collect 202. If N2 is the controller, then it will collect 101 samples, and N2 will collect 50.

This example demonstrates a useful way to design a schematic with multiple sinks. Choose one sink to control the simulation, and set all other sinks' ControlSimulation parameters to NO. In this manner, your sinks will collect appropriate amounts of data according to the multirate characteristics of your schematic.

Sources

Sources are components with no inputs. Sources that read from files, instruments, and data sets also have a ControlSimulation parameter. By default, its value is NO. When a source is controlling the simulation, it will keep the simulation running long enough to output all its data. Controlling sources can be used to create designs that process all the data in a file, as shown in the next figure.



In this example, the SDFRead component is controlling the simulation, and the TimedSink parameter is not in control. The TimedSink will collect all the data available in the file. This example demonstrates another useful way to design schematics: control the simulation with a source, and set all the sinks' ControlSimulation parameters to NO.

In the example, if both components' ControlSimulation parameters were flipped so that only the TimedSink was in control, then it would collect enough data to meet its Start and Stop parameters. If that were more data than was available in the file, then the SDFRead component would repeat its data or zero pad according to its Periodic parameter. If that were less data than was available in the file, then the SDFRead would not output the entire file.

It's possible to set both components' ControlSimulation parameters to YES. In that case, and if the file had more data than the TimedSink's Start and Stop required, then the SDFRead *would* output the entire file, but the TimedSink would ignore any data received after its Stop.

Multiple DF Controllers on the Schematic Are Not Allowed

Previous versions of Agilent Ptolemy (before ADS 1.5) allowed multiple Data Flow controllers on the same schematic, but this is no longer possible. Multiple controllers

were used to simulate the same design with different Data Flow parameters, for example with a different value of DefaultNumericStart. You can achieve the same effect by using single-point sweeps on the parameter you are interested in varying.

Agilent Ptolemy Components

The component libraries available for use with signal processing designs using the Agilent Ptolemy simulator are shown below. Reference information for each component is available by choosing Help, either from the parameters dialog box for a specific component, or from the Help menu.

Get to know the available components by choosing *Insert > Component > Component Library*, resizing the dialog box so you can read the complete names, and browsing through the list. [Table 3-2](#) summarizes the libraries and their contents.

Table 3-2. Agilent Ptolemy Component Libraries

Library	Summary of Contents
Antennas & Propagation	Contains components dealing with the radio channel, including antennas and propagation models. The channel models provide built-in functionality based on various standards, e.g., GSM, TDMA, CDMA.
Circuit Cosimulation	Contains items used to set up cosimulation with analog/RF circuits.
Common Components	A factory list of the most commonly used components taken from the remainder of the libraries.
Controllers	Items that control simulation parameters.
HDL Blocks	Components needed for HDL cosimulation.
Instruments	Contains components used to link data to instruments, such as the Agilent 89400 Vector Signal Analyzer.
Interactive Controls and Displays	Components that control and interactively display real-time simulation results. Data is not saved.
Numeric Communications	Contains numeric communications functions, such as ADPCM coder, QAM encoder, Viterbi decoder, modulation, demodulation, scrambler, spreader.
Numeric Control	Items that manipulate the flow of data during simulation, such as commutators, multiplexers, demultiplexers, upsamplers, forks, etc.
Numeric Logic	Contains Boolean operators, such as and, or, equals, greater than, etc.
Numeric Math	Contains math functions, such as adders, multipliers, integrators, log, sine, cosine, etc.

Table 3-2. Agilent Ptolemy Component Libraries (continued)

Library	Summary of Contents
Numeric Matrix	Contains components that receive and/or produce vector or matrix signals at their input and output, such as add and multiply. Also contains MATLAB components and components used for converting scalar to matrix.
Numeric Signal Processing	Contains basic discrete-time DSP functions, such as FIR filter, IIR filter and adaptive filter, DTFT, etc.
Numeric Sources	Contains sources (items having only output) that produce numeric signals. This includes sources that output scalar, matrix, and random signals.
Numeric Special Functions	Miscellaneous items. Typically nonlinear operations such as quantizing, limiting, or triggering on input signals.
Numeric Synthesizable DSP	Bit-accurate DSP models (adders, registers, etc.) with behavioral C++ simulation code and Verilog and VHDL synthesizable code.
Signal Converters	Converts signal (data) types, from one type to another, for example, CxToFloat (complex to float). Others include integer, fixed, or timed.
Sinks	Data collection items or data processed as measurements, such as numeric sink, BER sinks, or EVM sink.
Timed Data Processing	Contains data processing components that operate on time-domain baseband waveforms, e.g., multilevel symbol coders and converters, IQ data coders.
Timed Filters	Contains time-domain lowpass and bandpass analog filters for filtering baseband or RF signals.
Timed Linear	Contains various linear operations for time-domain analog baseband and RF signals, e.g., waveform delay, split, sum, sample, switch.
Timed Modem	Contains analog RF modulators, demodulators, and carrier recovery for AM, FM, PM, QAM, QPSK, GMSK, MSK, DQPSK, and Pi/4 DQPSK formats.
Timed Nonlinear	Contains various nonlinear time-domain operations for time-domain analog baseband and RF signals, e.g., nonlinear gain, RF mixers, RF multipliers, rectifiers, signal sampling, or phase detectors.

Table 3-2. Agilent Ptolemy Component Libraries (continued)

Library	Summary of Contents
Timed RF Subsystems	Contains various RF subsystem components, such as RF combiner, RF modulator, or RF demodulator.
Timed Sources	Contains various time-domain signal generators for baseband and RF signals, e.g., AM, FM, PM, QAM, clock, sinusoid, pulsed, or video.

Note If you have purchased and installed ADS Design Library products, such as the CDMA, GSM, W-CDMA, cdma2000, EDGE, DTV, 1xEV, WLAN, or W-CDMA3G design libraries, they will be displayed in the list, in alphabetical order.

Chapter 4: Understanding Parameters

Value Types

Agilent Ptolemy requires specific parameter value types (such as string, real array, or complex) for the component parameter values you enter in schematic designs. They are called value types.

Component parameter values may be entered several ways by:

- Editing the Component Parameters dialog box. Double-click the component symbol on the schematic, a dialog box appears. Parameter values may be selected from lists or entered. The dialog box lists the value type expected, such as real or integer.
- Editing values directly on the schematic. Click the parameter value and type.
- Editing default values in the Design Definition dialog box. Choose *File > Design / Parameters > Parameters tab*. A type of parameter value can be selected from the Value Type list, and a default value can be entered in the Default Value field.

This section describes these value types. [Table 4-1](#) lists each value type and its use:

Table 4-1. Ptolemy Parameter Value Types

Value Type	Description
Real	Editing in Component Parameter dialog box: A. Enter real number. B. Enter expression for a real value—Example: $X*\cos(Y)$, where X and Y are defined expressions. Parameter editing on schematic: Highlight parameter value on schematic and enter real value or expression.
Integer	Editing in Component Parameter dialog box: A. Enter integer. B. Enter expression for an integer value—Example: $X+Y$, where X and Y are defined expressions. Parameter editing on schematic: Highlight parameter value on schematic and enter integer value or expression.

Table 4-1. Ptolemy Parameter Value Types (continued)

Value Type	Description
Fixed Point	<p>Parameter editing in Component dialog box:</p> <p>A. Enter real value, but the value used will be based on the precision used with this parameter.</p> <p>B. Enter expression for a real value—Example: $X \cdot \cos(Y)$, where X and Y are defined expressions.</p> <p>Parameter editing on schematic: Highlight parameter value on schematic and enter real value or expression.</p>
Complex	<p>Editing in Component Parameter dialog box:</p> <p>A. Enter a complex number using the form $Re + j * Im$.</p> <p>B. Enter expression for a complex value—Example: $\cos(X) + j \cdot \sin(Y)$, where X and Y are defined expressions, j is the imaginary operator.</p> <p>Parameter editing on schematic: Highlight parameter value on schematic and enter complex value or expression.</p>
String	<p>Editing in Component Parameter dialog box:</p> <p>A. Enter string. <i>Do not</i> enclose this string with any double quote symbols. Note for embedded double quotes ("), use double double quotes ("").</p> <p>B. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for a string value.</p> <p>Parameter editing on schematic: Highlight parameter value on schematic and enter string value enclosed with double quote symbols.</p>
Precision	<p>Editing in Component Parameter dialog box:</p> <p>A. Enter string in the form X.Y or Y/W. <i>Do not</i> enclose this string with any double quote symbols. The form X.Y, such as 8.24, means that there are X bits (including sign bit) to the left of the decimal point, and Y bits to the right of the decimal point. The form Y/W, such as or 24/32, means that there are Y bits to the right of the decimal point and W bits total. Note that $X+Y=W$.</p> <p>B. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for a precision value.</p> <p>Parameter editing on schematic: Highlight parameter value on schematic and enter string value enclosed with double quote symbols.</p>

Table 4-1. Ptolemy Parameter Value Types (continued)

Value Type	Description
Filename	<p>Editing in Component Parameter dialog box:</p> <p>A. Enter string for the name of a file including the pathname and select an extension type. The filename may include environment variables such as <code>-/</code>, <code>\$HOME</code>, <code>\$HPPEESOF_DIR</code>, or others.</p> <p>B. Enter value by reference—Example: <code>@Y</code>, where <code>Y</code> is a the name of a Variable or Expression for a filename value.</p> <p>Parameter editing on schematic: Highlight parameter value on schematic and enter string value enclosed with double quote symbols.</p>
Integer Array	<p>Editing in Component Parameter dialog box:</p> <p>A. Enter integer values directly—Example: <code>1 3 -2 5</code> (spaces separate data).</p> <p>B. Enter values from a file—Example: <code><filename</code>. If the filename has no path specified, the project data directory is used. The content of the file must be numbers separated by spaces or on a new line. For example:</p> <pre>1 -2 5 2 and 1 -2 5 2</pre> <p>are equivalent.</p> <p>C. Enter values directly in addition to file data—Example: <code>1 <file1 2</code>. If <code>file1</code> contains <code>-2 5</code>, then the array would be the same as in A.</p> <p>D. Enter value by reference—Example: <code>@Y</code>, where <code>Y</code> is a the name of a Variable or Expression for an integer array.</p> <p>Parameter editing on schematic: Highlight parameter value on schematic and enter array value enclosed with double quote symbols.</p>

Table 4-1. Ptolemy Parameter Value Types (continued)

Value Type	Description
Fixed Point Array or Real Array	<p>Editing in Component Parameter dialog box:</p> <p>A. Enter fixed-point values directly—Example: 1.2 -2.3 5.6 2.8 (spaces separate data).</p> <p>B. Enter values from a file—Example: <filename. If the filename has no path specified, the project data directory is used. The content of the file must be numbers separated by spaces or on a new line. For example: 1.2 -2.3 5.6 2.8 and 1.2 -2.3 5.6 2.8 are equivalent.</p> <p>C. Enter values directly in addition to file data—Example: 1.2 <file1 2.8. If file1 contains -2.3 5.6, then the array would be the same as in A.</p> <p>D. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for a fixed-point or real array.</p> <p>Parameter editing on schematic: Highlight parameter value on schematic and enter array value enclosed with double quote symbols.</p>

Table 4-1. Ptolemy Parameter Value Types (continued)

Value Type	Description
Complex Array	<p>Editing in Component Parameter dialog box:</p> <p>A. Enter complex values directly as ordered pairs separated by a comma— Example: (1.2, 2.5) (-2.3, 1.3) (5.6, -1.4) (2.8, 3.4)(each ordered pair is enclosed with parentheses, spaces separate each ordered pair of data).</p> <p>B. Enter values from a file—Example: <filename. If the filename has no path specified, the project data directory is used.The content of the file must be numbers separated by spaces or on a new line. For example: 1.2 2.5 -2.3 1.3 5.6 -1.4 2.8 3.4 and 1.2 2.5 -2.3 1.3 5.6 -1.4 2.8 3.4 are equivalent.</p> <p>C. Enter values directly in addition to file data—Example: (1.2,2.5) <file1 (2.8,3.4). If file1 contains -2.3 1.3 5.6 -1.4, then the array would be the same as in A.</p> <p>D. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for a complex array.</p> <p>Parameter editing on schematic: Highlight parameter value on schematic and enter array value enclosed with double quote symbols.</p>

Table 4-1. Ptolemy Parameter Value Types (continued)

Value Type	Description
String Array	<p>Editing in Component Parameter dialog box:</p> <p>A. Enter string values directly—Example: "Button 1" "Button 2" "Button 3"(each string is enclosed with double quote marks, spaces separate each string).</p> <p>B. Enter values from a file—Example: <filename. If the filename has no path specified, the project data directory is used. The content of the file must be text separated by spaces or on a new line. For example: "Button 1" "Button 2" "Button 3" and "Button 1" "Button 2" "Button 3" are equivalent.</p> <p>C. Enter values directly in addition to file data—Example: "Button 1" <file1 "Button 3". If file1 contains "Button 2," then the array would be the same as in A.</p> <p>D. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for a string array.</p> <p>Parameter editing on schematic: Highlight parameter value on schematic and enter array value enclosed with double quote symbols.</p>
Enumerated Type (with form specific to the component)	<p>Editing in Component Parameter dialog box:</p> <p>A. Select enumerated type from selection list specific to the component parameter. For example, Time Unit (milliseconds, etc.) is an Enumerated Type you choose from a list. The Data Flow Controller parameter Scheduler Type is also an Enumerated Type.</p> <p>B. Select the "Standard" enumerated type and enter an integer value in the entry field provided. The integer value is associated with an option in the selection list with the first selection list entry associated with the integer 0, the second entry with the integer 1, etc.</p> <p>C. Select the "Standard" enumerated type and enter the expression in the entry field provided for an integer value—Example: X+Y, where X and Y are defined expressions.</p> <p>Parameter editing on schematic: Highlight parameter value on schematic and enter enumerated value, or use the up or down arrow keys on the keyboard to scroll through the enumerated options available.</p>

All of the above parameter types, except the Enumerated type, are directly available to define parameters in a schematic design.

To define or see the list of value types for a schematic design, from a Signal Processing Schematic window, choose *File > Design / Parameters > Parameter*. The list of parameter types available for a schematic design can be seen by selecting the Value Type drop-down list.

To use the Enumerated type for a schematic design, you must edit the design AEL file (located in the project networks directory) and implement the AEL code for the desired Enumerated type. Examples of AEL for Enumerated types can be observed in the file: \$HPPEESOF_DIR/hptolemy/ael/stars.ael.

Parameter Editing

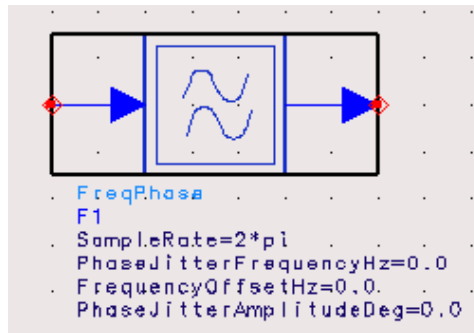
When an instance of a component or design is placed on the schematic, its parameters can be viewed below its symbol.

The default is for parameters to be visible on the schematic. To enable parameter visibility on the schematic, check two areas. From the Schematic window, choose *Options > Layers*, a dialog box appears. In the Layers list (left), select *Parameters*. Make sure that the *Visible* box is checked (center). Next double-click any component in the schematic. This displays the component parameters dialog box. Make sure that the *Display* parameter on schematic box (lower-center) is checked.

To illustrate this procedure, we will place an instance of the FreqPhase component.

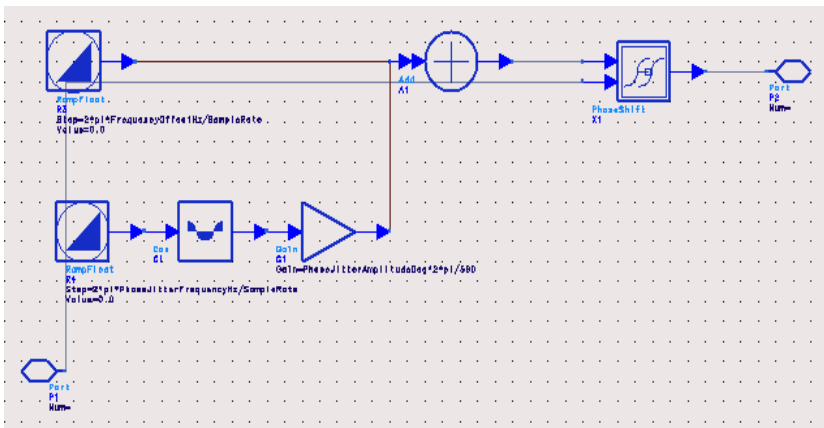
1. Choose **Insert > Component > Component Library**.
2. From the Library list, select **Numeric Communications** and then **FreqPhase** from the Components list on the right.
3. Next place this component on the schematic.

Observe that its parameters (visible below its symbol on the schematic) are SampleRate, PhaseJitterFrequencyHz, FrequencyOffsetHz, and PhaseJitterAmplitudeDeg, as shown in the figure below. This component is actually a schematic design that we will look at in more detail.:



Notice that each parameter has a numeric value. These values could just as easily be an expression. Note that the SampleRate parameter is given as an expression, $2*\pi$. The full features of Advanced Design System expressions are discussed in the *Expressions, Measurements, and Simulation Data Processing* manual.

To observe the detail of the schematic design associated with the FreqPhase component, click the symbol to highlight it, then choose *View > Push into Hierarchy* or click the *Push into Hierarchy* button (down arrow icon) from the toolbar. When this is done for FreqPhase, the schematic shown below appears..



Notice that the various components in this schematic reference the top-level FreqPhase parameters by name. The top RampFloat component has its Step parameter equal to $2*\pi*FreqOffsetHz/SampleRate$, where FreqOffsetHz and SampleRate are value passed in from the top level.

To observe the parameters defined (or to define additional parameters) for this schematic design, choose *File > Design / Parameters*. The Design Definition dialog box appears. Select the *Parameters* tab and you will observe fields to enter parameter definition information. For more information on the Design Definition dialog box refer to Chapter 4, *Creating Hierarchical Designs in the User's Guide*.

Parameter Expressions

Parameter values can be arithmetic expressions. This is particularly useful for propagating values down from a top-level system parameter to component parameters down in the hierarchy. An example of a valid parameter expression is:

```
x = pi/(2*order)
```

where `order` is a parameter defined in the network or top-level system, and `pi` is the built-in constant π . The basic arithmetic operators are addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^). These operators work on integers and floating-point numbers. Currently, all intermediate expressions are calculated in double-precision values and only the final value is converted to the type of the parameter being computed. Hence, it is necessary to be very careful when, for example, using floating-point expressions, to compute an integer parameter. In an integer parameter specification, all intermediate expressions will be calculated with double-precision floating-point values and the final value is cast to an integer value.

Complex-Valued Parameters

When defining complex values, the basic syntax is

```
real + j*imag
```

where `real` and `imag` evaluate to double-precision, floating-point values, which may be numbers or expressions, and where `j` is the imaginary operator.

There are also other functions in Agilent Ptolemy that can be used with complex values. These include:

- An expression/function that returns a Cartesian form: `complx (X, Y)`.
- An expression/function that converts a polar form to Cartesian form: `polar (X, Y)`, where `X` is magnitude and `Y` is in degrees.
- An expression/function that converts a decibel form to a Cartesian form: `dbpolar (X, Y)`, where `X` is in decibels and `Y` is in degrees.

Parameters for Fixed-Point Components

Many of the fixed-point components used in Agilent Ptolemy utilize one or more common parameters that identify the specific characteristics of the finite-precision, fixed-point value. These include characteristics specifying overflow, overflow reporting, quantization, finite precision bit format, and more. The following describes several properties in common use by these components.

1. Parameters specifying fixed-point value precision are typically labeled “Precision,” “InputPrecision,” “OutputPrecision,” or some other token containing “Precision.”

Fixed-point parameter precision is defined by either of two types of syntax:

Syntax 1

As a string such as “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 $m+n$.

Syntax 2

A string like “24/32” which means 24 fraction bits from a total word length of 32. This format, n/w , 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. The maximum value of w (or $x+y$) is 256.

Thus, for example, a fixed-point value of 0.8 may have a precision defined as 2/4. This means that a 4-bit word will be used with two fraction bits. Since the value “0.8” cannot be represented precisely in this precision, the actual value of the parameter will be rounded to “0.75.”

When an input pin with an associated fixed-point signal class (scalar or matrix) receives another class of signal (scalar or matrix, respectively), the received signal is automatically converted to the fixed-point class. A pin specified for use with fixed-point scalar signals does not accept any matrix class signals, and vice versa. The automatic conversion from timed, complex or floating-point signals to a fixed-point signal uses a default bit width of 32 bits with the minimum number of integer bits needed to represent the value. For example, the automatic conversion of the float value of 1.0 creates a fixed-point value with precision of 2.30, and a value of 0.5 would create one of precision of 1.31. For details on data/signal conversion rules, refer to [“Conversion of Data Types” on page 5-5](#).

2. The parameter used to specify the arithmetic form of a fixed-point value is labeled `ArithType`. This parameter is an enumerated type with two options: `TWOS_COMPLEMENT` and `UN_SIGNED`. The fixed-point components in the Numeric Synthesizable DSP library may use either arithmetic form. However, fixed-point components outside the Numeric Synthesizable DSP library may only use the `TWOS_COMPLEMENT` form, which is the default.
3. The parameter used to specify the quantization property of a fixed-point value is labeled `RoundFix`. This parameter is an enumerated type with two options:

ROUND and TRUNCATE. The quantization property is used to convert a floating-point value to its fixed-point value. The ROUND quantization property causes this float-to-fixed transformation to occur such that the nearest fixed-point value to the floating-point value is used. For example, consider the floating-point value 0.1. It is not possible to represent this number exactly as a two's complement fixed-point value. Remember that a fractional decimal number is represented in its fixed-point form by composing it of the summation of fractional powers of two (2^{-N}). 0.1 is represented as 0.0001100110011...with an infinite number of fractional binary terms. If the precision is 2.8 and the quantization is ROUND, then this above fixed-point value is rounded up to the nearest fractional power of 2^{-8} which is 0.00011010. If the precision remains at 2.8 and the quantization is TRUNCATE, then the value is truncated to 0.00011001.

4. The parameter used to specify the overflow properties of fixed-point mathematical operations is labeled OverflowHandler or OvflwType. Both of these are an enumerated type. The OverflowHandler parameter has four options: wrapped, saturate, zero_saturate, or warning. The OvflwType parameter has two options: wrapped or saturate. The OvflwType parameter is used only by fixed-point components in the Numeric Synthesizable DSP library. The OverflowHandler is used by all other fixed-point components. The overflow parameter specifies the overflow characteristic to use when the result of a fixed-point operation cannot fit into the precision specified.

Consider a fixed-point ramp data source (RampFix) as shown in the following figure. It has a step size of 0.2, initial value of 0, output precision of 2.14, with round type quantization.

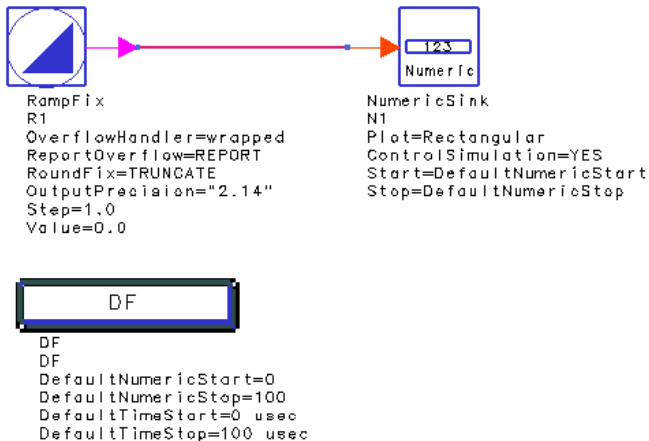


Figure 4-1. Schematic Using the RampFix Component

When the OverflowHandler is set to *wrapped*, the following data display results:

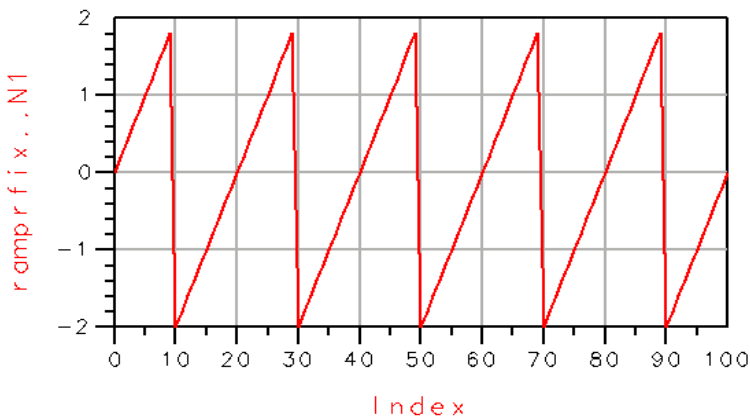


Figure 4-2. Simulation Plot with Overflow Handler Set to Wrapped

Note that as a 2's complement signal, the maximum value for a 2.14 precision with rounding is nearly 1.9 and the minimum value is nearly -2.0. There are actually more decimal places in these values due to the quantization of the step size. This

maximum and minimum is obtained by first converting the step size of 0.2 into its fixed-point form with 2.14 precision. This becomes the step size for the fixed-point ramp accumulation. The signal begins at zero, and increments by the fixed-point binary representation of 0.2 with each sample. When the maximum value is reached, the output wraps to the minimum value for the given precision and quantization.

When the above example uses the *truncate* type of quantization the following data display results:

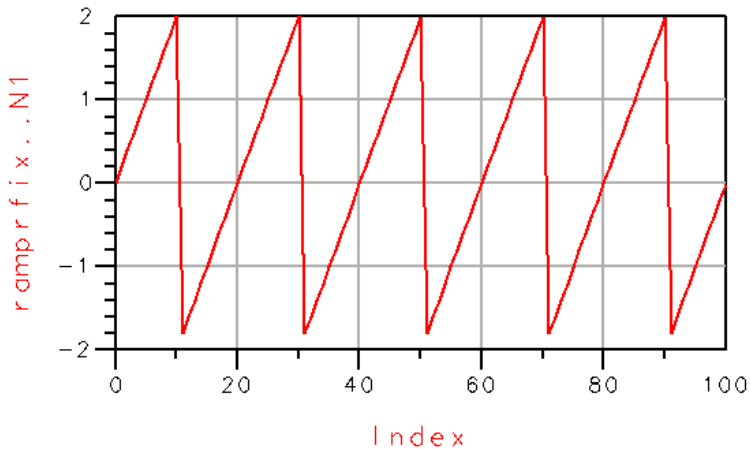


Figure 4-3. Simulation Plot with Truncate Quantization

Note that as a 2's complement signal, the maximum value for a 2.14 precision with truncation is 2.0, and the minimum value is -1.9. The signal begins at zero, and increments by 0.2 with each sample. When the maximum value is reached, the output wraps to the minimum value for the given precision.

With truncation used, but with overflow set to *saturate*, the following data display results:

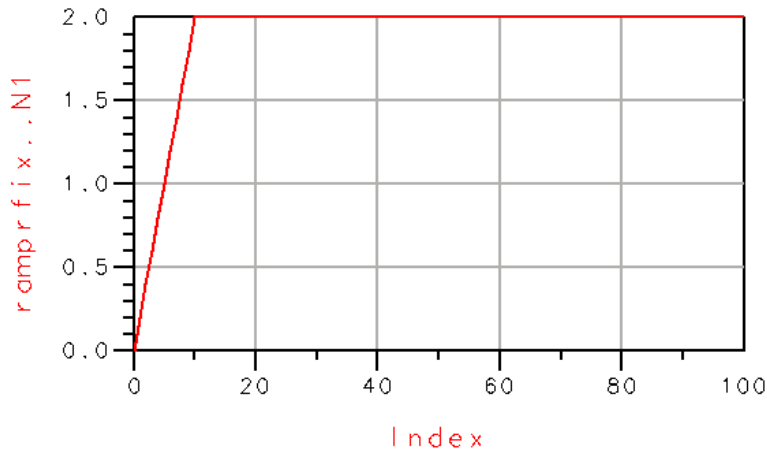


Figure 4-4. Simulation Plot with Truncate Quantization and Overflow Handler Set to Saturate

Note that when the ramp rises to 2.0, it stays constant at that level.

With truncation used, but with overflow set to *zero_saturate*, the following data display results:

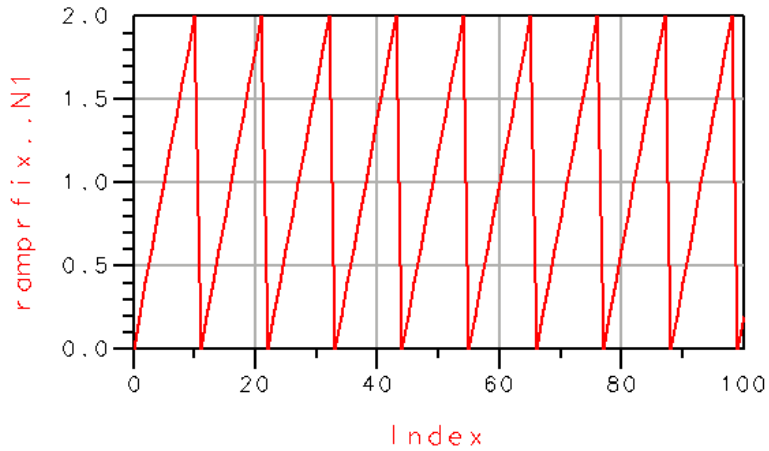


Figure 4-5. Simulation Plot with Truncate Quantization and Overflow Handler Set to Zero_Saturate

Note that when the ramp rises to 2.0, it resets to the value of zero and continues to rise.

Again with truncation used, but with overflow set to *warning* the following data display results:

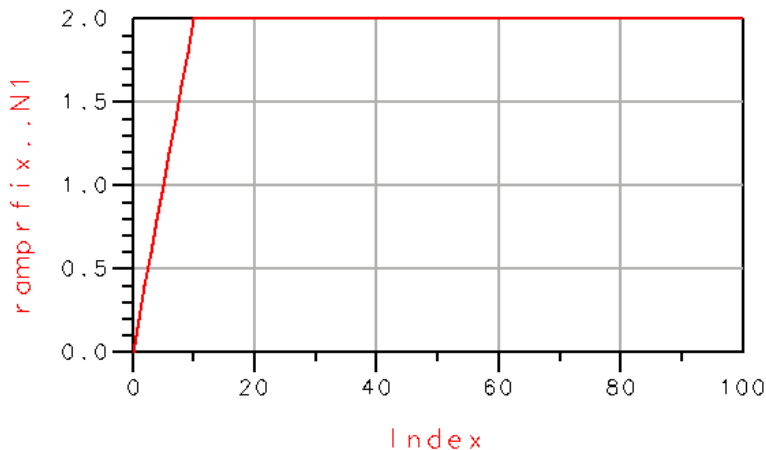


Figure 4-6. Simulation Plot with Truncate Quantization and Overflow Handler Set to Warning

Note that the saturate characteristic is used. Additionally, the warning mode results in ReportOverflow (explained below) being set to REPORT. This reports the number of overflows at the end of the simulation.

5. The parameter used to specify whether overflow is reported is labeled ReportOverflow. This parameter is an enumerated type with two options: REPORT and DONT_REPORT. Consider the preceding overflow displays. For each of the cases, when the ReportOverflow parameter is set to REPORT a warning message is displayed in the Simulation/Synthesis Messages window after simulation. In the previous simulation for the *zero_saturate* data display, the warning is:

1: R1: experienced overflow in 9 out of 102 fixed-point calculations checked (8.8%)

When you click this message, the RampFix component with the name R1 is highlighted in the schematic window.

When the ReportOverflow parameter is set to DONT_REPORT, this warning message does not appear.

- The parameter used to specify whether a component is to use the input signal with its arriving precision, or whether this signal is to be cast into another component's specific precision is labeled `UseArrivingPrecision`. This parameter is an enumerated type with two options: `NO` or `YES`. This parameter is used with the `InputPrecision` parameter. When `UseArrivingPrecision = NO`, the input signal is cast to the precision specified by `InputPrecision`. Otherwise, the input signal's precision is used.

String Parameters

String parameters are assigned a text value that may include any alpha-numeric symbol, including spaces and other punctuation symbols. If a double-quote symbol (“”) is to be used, it must be used with two such sequential symbols (“”) and will be interpreted as only a single, double-quote symbol.

Filename Parameters

Filename parameters are assigned a filename value that may include the file path name and environmental variables such as `~/`, `$HOME`, `$HPEESOF_DIR`, or others. If no path name is provided, the current project data subdirectory is the assumed path for the file.

Array Parameters

When defining arrays of integers, floating-point numbers, complex numbers, fixed-point numbers, or strings, the basic syntax is a simple list separated by spaces, as shown in the following example:

```
1 2 3 4 5
```

defines an integer array with five elements. Repetition can be indicated using the following syntax, `value[n]`, as demonstrated in the following example:

```
1 2 3[10] 4 5
```

where `n` is an integer. This example has ten instances of the value 3. An array or portion of an array can be input from a file using the symbol “<” as shown in the following example:

```
1 2 < filename 3 4
```

Here the first two elements of the array will be 1 and 2, the next elements will be read from file filename, and the last two elements will be 3 and 4. This latter capability can be used in combination with the WaveForm component to read a signal from a file.

When defining complex array values, the basic syntax is

```
(1.2, 2.5) (-2.3, 1.3) (5.6, -1.4) (2.8, 3.4)
```

where all entered complex values are ordered pairs of real and imaginary values of complex numbers enclosed in parentheses and all entries must be numbers.

When defining string array values, the basic syntax is

```
“Button 1” “Button 2” “Button 3”
```

where all entered string values are enclosed in double quote symbols.

Reading Array Parameter Values From Files

The values of all array parameter types can be read from a file. The syntax for this is to use the symbol “<” as in the following example:

```
< filename
```

```
or
```

```
1.2 2.6 <filename 2.8 6.4
```

If the filename has no path specified, the project data directory is used. Otherwise, the filename should typically contain the full pathname to the file. Any references to environment variables or home directories are substituted to generate a complete path name. All values in the filename must be numeric values for the numeric array types (integerarray, realarray, fixedpointarray, complexarray), and must be string values for the string array type. The contents of the file are read and spliced into the parameter expression and re-parsed. File inputs can be very useful for array parameters which may require a large amount of data. Other expressions may come before or after the “< filename” syntax (any white space that appears after the < character is ignored). Within the file, comment lines containing a leading pound (#) symbol are ignored by the file parser.

Parameters With Optimization and Swept Attributes

Many component parameters may have associated attributes that are used during nominal optimization. Within the Component dialog box, any parameter of type Real, FixedPoint, Integer, or Enumerated, may also be optimized for design performance. A complex value may be optimized by optimizing its real and/or imaginary parts.

Parameters of type Complex, Precision, Array, String, or Filename may be optimized or swept by creating a string that references optimized or swept variables. To reference an optimized variable, the variable must be defined in a VAR (Variables and Equations) component with the Standard entry mode and with Optimization/Statistic Setup enabled.

In a similar manner to optimization attributes, there can also be parameters with swept attributes.

For more information on optimization in Agilent Ptolemy, refer to [Chapter 8, Using Nominal Optimization](#) in this manual. For more information on sweeping parameters in Agilent Ptolemy, refer to [Chapter 7, Performing Parameter Sweeps](#) in this manual.

Chapter 5: Using Data Types

This chapter reviews some basic material on Data Types that was introduced in [Chapter 3, Data, Controllers, Sinks, and Components](#), but then goes into more detail.

Agilent Ptolemy uses different data types such as integer, fixed-point, floating-point, and complex in scalar or matrix forms. In Agilent Ptolemy documentation there are numerous references to data and signal types. When data is presented versus an independent variable such as time, the data can be thought of as a signal. Regardless of the terminology, data or signals consist of packets of information that are passed from one component to another.

Representation of Data Types

Agilent Ptolemy schematics contain component stems with different colors and thicknesses. Each component input and output pin has an associated data type, and each type is represented in the component symbol by use of a color code and a thickness of stem. Additionally, each component stem may have single or multiple arrowheads. [Table 5-1](#) describes stem color and thickness in Signal Processing schematics.

Table 5-1. Ptolemy Component Stem Color and Thickness

Data Type	Stem Color	Stem Thickness
Scalar Fixed Point	Magenta	Thin
Scalar Floating Point	Blue	Thin
Scalar Integer	Orange	Thin
Scalar Complex	Green	Thin
Matrix Fixed Point	Magenta	Thick
Matrix Floating Point	Blue	Thick
Matrix Integer	Orange	Thick
Matrix Complex	Green	Thick
Timed	Black	Thin
AnyType	Red	Thin

Stem Thickness

Figure 5-1 illustrates stem thickness:

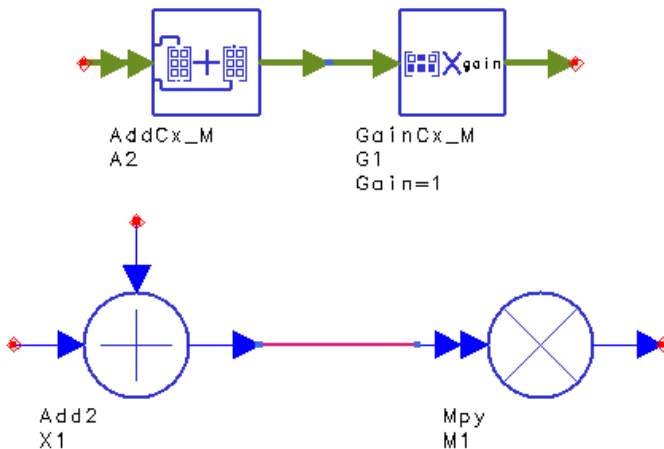


Figure 5-1. Matrix Data (Thick Lines) vs. Scalar Data (Thin Lines)

Single and Multiple Arrowheads

Agilent Ptolemy uses block diagram schematics to enter information for simulation, which implies that all signals flowing between components are directional. Therefore, each input or output stem has arrowheads indicating the signal flow direction. This is not the case in circuit schematics where signals (wires) are generally bidirectional.

The signal flow is indicated by a single arrowhead. While single arrowhead stems carry only one distinct signal, double arrowhead stems can carry any number of independent signals or data. Figure 5-2 shows the difference between single and

multiple arrowheads. In this figure, the input of the multiplier component is a single multiple input carrying data from any number of inputs.

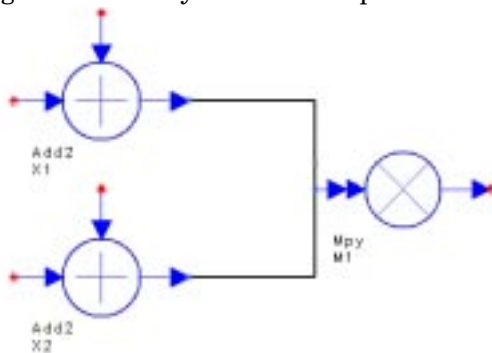


Figure 5-2. Single and Multiple Arrowheads

Data Types Defined

Presently, there are two general signal types used in Agilent Ptolemy, *numeric* and *timed*. The numeric type has several subtypes, such as fixed-point, real, scalar, and matrix. Numeric signals have sequential numbers as the independent variable. Timed signals have time as the independent variable and are derived from complex data. Timed signals have additional attributes.

Typically, numeric data is used for algorithmic development in the baseband portion of a communication system. Timed signals are used to simulate the signal in the modulation channel, as well as for cosimulation with certain Advanced Design System circuit simulators.

Numeric Scalar Data

Numeric scalar data is defined as follows:

- **int**—single, integer value (signed value defined with a 32-bit value)
- **fixed**—single, fixed-point value with the following properties and operation attributes:

precision defined using $x.y$ or y/w where

x = bits to the left of the decimal point

y = bits to the right of the decimal point

$w = x+y =$ total bit width, 1 to 255

arithmetic type

two's complement (with sign bit included in x)

unsigned

quantization type

truncate, round

overflow type

saturate, saturate to zero, wrapped

- float—double precision floating point number
- complex—pair of double precision floating point number for real and imaginary parts

Numeric Matrix Data

All matrix data is defined as a two-dimensional array (rows, columns) of either int, fixed, float, or complex values. All matrix data types are indicated by thick stems, in contrast with the thin stems used for scalar data types.

Timed Data

Agilent Ptolemy supports timed data. This signal is derived from complex data and includes additional attributes. The timed signal packet includes five members

$\{i(t), q(t), \text{flavor}, Fc \text{ and } t\}$

where $i(t)$ and $q(t)$ are the timed signal in phase and quadrature components, *flavor* indicates the representation of a modulated signal, Fc 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 Fc can be converted to the real BaseBand flavor $V(t)$ as:

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

Conversion of Data Types

What Happens During Conversion?

We introduced this topic in Chapter 3. In this section, we go into more detail. Most conversions do what you expect. For example, when converting from lower precision to higher precision data types, such as integer to float, no data is lost; only the format is changed.

When converting from higher precision to lower precision data types, such as float to integer, the outcome is governed by your computer’s math rounding rules.

Whether you manually place a converter, or the simulator “splices” in a converter, the conversion process is the same. It is similar to the casting operation used in C or C++ languages. If the conversion from A to B requires more information (integer to float, float to complex, etc.) the “obvious” thing happens. For example, conversion from float to complex is done by setting the imaginary part of the complex number equal to 0.0. However if the conversion involves loss of information (complex to double, double to integer, etc.), a set of rules are followed that are in most cases very simple and intuitive.

Numeric Scalar and Matrix Conversions

Table 5-2 outlines the rules regarding scalar conversions among numeric data types:

Table 5-2. Numeric Scalar and Matrix Conversion Rules

From	To		
	<i>Integer</i>	<i>Fixed</i>	<i>Float</i>
<i>Complex</i>	round mag	round/truncate mag	mag
<i>Float</i>	round	round/truncate	
<i>Fixed</i>	round		

Note that *mag* in preceding table means the magnitude of the complex number $C = a + jb$, which is equal to

$$\sqrt{a^2 + b^2}.$$

For automatic conversion (when no converter is explicitly used) to the Fixed data type, the resulting fixed-point number has the default length of 32 bits and a precision of the minimum number of integer bits needed for a two's complement representation. For example, the integer 5 is converted to the fixed point number 0101.0000000000000000000000000000 (precision "4.28"), whereas the float number 3.375 is converted to 011.0110000000000000000000000000 (precision "3.29"). If this is not the behavior you want, you must explicitly use a converter.

For matrix conversions, the above operations hold for all entries in the matrix.

Timed Data Conversions

You can convert between timed and scalar numeric data types by placing one of the following converters and supplying the parameters as needed:

- Timed to Complex or Complex to Timed
- Timed to Float or Float to Timed
- Timed to Fixed or Fixed to Timed
- Timed to Integer or Integer to Timed

Given the Timed data type $\{i(t), q(t), \text{flavor}, F_c \text{ and } t\}$, the conversions between input and output of a converter are summarized below:

Timed To Float



If x is the input and y is the output for the TimedToFloat converter, then:

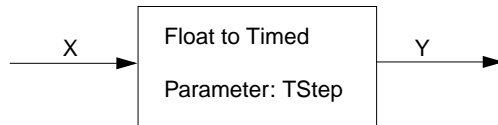
$$y[n] = i(t)\cos(2\pi F_c t) - q(t)\sin(2\pi F_c t)$$

when `flavor = ComplexEnv`

$$y[n] = i(t)$$

when `flavor = BaseBand`

Float To Timed



The FloatToTimed converter has one specific parameter, TStep. If x is the input and y is the output for this converter, then the y(t) packet has the following parts:

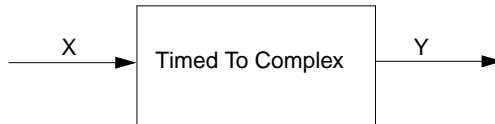
$$i(t) = x[n \cdot TStep]$$

$$q(t) = 0.0$$

$$F_c = 0.0$$

flavor = BaseBand

Timed To Complex



The Timed To Complex converter has no parameters. If x is the input and y is the output for this converter, then:

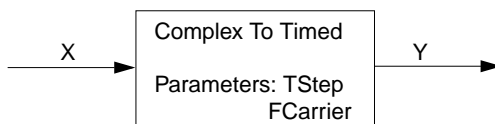
$$y[n] = i(t) + jq(t)$$

when flavor = ComplexEnv

$$y[n] = i(t) + j \cdot 0.0$$

when flavor = BaseBand

Complex To Timed



The Complex To Timed converter has two specific parameters, TStep and FCarrier. If x is the input and y is the output for this converter, then the $y(t)$ packet has the following parts:

$$i(t) = \text{Real}\{x[n \cdot T\text{Step}]\}$$

$$q(t) = \text{Imag}\{x[n \cdot T\text{Step}]\}$$

$$F_c = F\text{Carrier}$$

$$\text{flavor} = \text{ComplexEnv}$$

Rules and Exceptions

The converter devices are in general not reciprocal, i.e., putting two converters with opposite functionality back-to-back does not necessarily recover the original signal.

Based on the three categories of numeric scalar, numeric matrix, and timed data types discussed above, the following rules should be considered:

- The conversion between numeric scalar and numeric matrix types are done by explicitly placing Pack and UnPack components. No automatic conversion is performed between these two categories.
- The conversion between numeric scalar and timed data is done by placing the appropriate converters. Automatic conversion between these two categories is allowed (see details below).
- There is no direct conversion between numeric matrix and timed data types.

Figure 5-3 summarizes the conversion among data types.

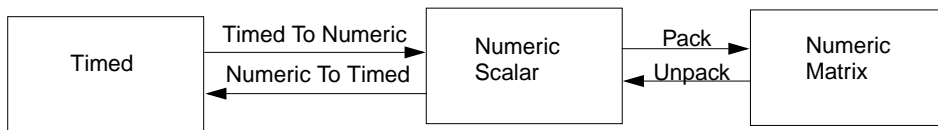


Figure 5-3. Timed Data Must be Converted to Numeric Scalar Before Being Converted to Matrix

Automatic or Manual Data Type Conversion

If the output of component A and the input of component B is the same (which means they are represented by the same color), the data is simply copied from A to B. If the

output of component A and the input of component B is different, a conversion is necessary.

Although the software will automatically convert dissimilar data types, such as complex to float, you may want to manually place an appropriate converter (from the Signal Converters library) in your schematic. This acts as a visual reminder that a conversion is taking place, and also helps you decode error messages that may arise. Automatic conversion means that an appropriate converter is “spliced in” behind the scenes and is not shown on the schematic.

Automatic conversion *is* allowed *among* scalar data types and *among* matrix data types, but *not between* scalar and matrix data types.

Allowed and Disallowed Automatic Conversions

Automatic conversion is available among all numeric scalar types. The same is true for matrix types. [Figure 5-4](#) summarizes the allowed conversions.

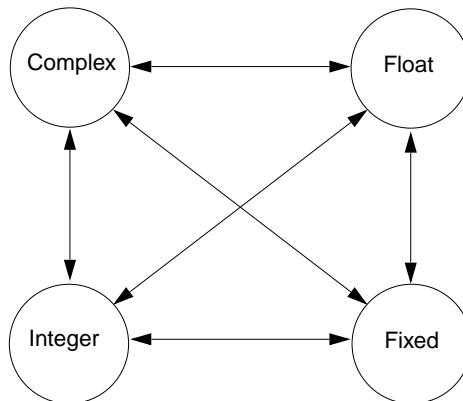


Figure 5-4. Automatic Conversion Among Numeric Scalar and Matrix Types

With one exception (complex to timed), automatic conversion between timed and numeric scalar types is also supported, as depicted below in [Figure 5-5](#).

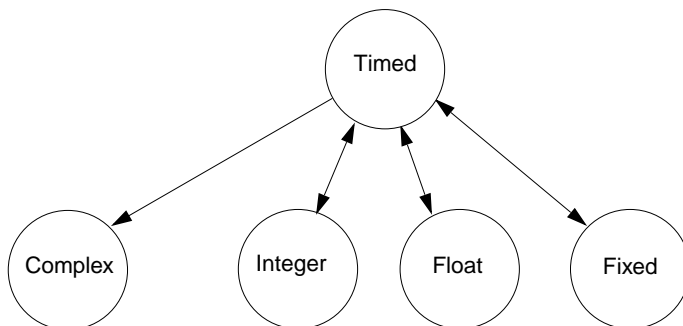


Figure 5-5. Automatic Conversion of Timed and Scalar Numeric Types

Automatic conversion from Complex to Timed is *not* supported because the information about the carrier frequency must be supplied by the user. You need to place a ComplexToTimed converter and enter the appropriate parameters if such a conversion is needed.

Also note that automatic conversion of Float to Timed, Fixed to Timed, Integer to Timed, or Complex to Timed is only possible when there is at least one component in the design defining the TStep.

When a scalar pin is directly connected to a matrix pin (or vice versa), without a Pack or Unpack converter, an error message is generated.

In the Numeric Matrix Library, there are four converters, such as Pack_M and PackCx_M, that are used to “pack” scalar data into matrix data. Similarly, there are four converters, such as UnPk_M and UnPkCx_M, that “unpack” the data (back to scalar). There is no automatic conversion between scalar and matrix data (or vice versa). You must place the converters where needed in your design.

Chapter 6: Understanding File Formats

Real, complex, and string array data can be used with component parameters of type real array, complex array, or string array, respectively. Real array data can be used as input with the ReadFile component. Real, complex, floating matrix, fixed matrix, complex matrix, and integer matrix can be used as output from the Printer component.

In the following seven examples of Agilent Ptolemy file formats (from Real Array Data through Complex Matrix Data), the examples are drawn from the code and include the “#” symbol, which denotes a comment on that line.

Real Array Data

```
# Template for Agilent Ptolemy real data
# Each number separated by newlines
1
0
0
```

Complex Array Data

```
# Template for Agilent Ptolemy complex data
# Each complex value, (real, imag), separated by newlines
(1.0, 0.0)
(0.0, 0.0)
(0.0, 0.0)
```

String Array Data

```
# Template for Agilent Ptolemy string data
# Each string value enclosed on double-quote marks, "", and separated by
newlines
"text 1"
"text 2"
"text 3"
```

Float Matrix Data

```
# Template for Agilent Ptolemy real matrix data
# Each matrix data set separately listed with brackets around each row and
```

```
matrix
# Each matrix row separated by newlines
[[ 1.2, -2,    2 ]
 [ -2,  2.25, -2 ]]
[[ 2.5, -2.1,  3.2 ]
 [ -3.5, 2.4, -1.3 ]]
[[ 2.2, -2.4,  3.8 ]
 [ -2.5, 2,   -2.6 ]]
```

Fixed-Point Matrix Data

```
# Template for Agilent Ptolemy fixed-point matrix data
# Each matrix data set separately listed with brackets around each row and
matrix
# Each matrix row separated by newlines
[[ 1.2, -2,    2 ]
 [ -2,  2.25, -2 ]]
[[ 2.5, -2.1,  3.2 ]
 [ -3.5,  2.25, -1.25 ]]
[[ 2.2, -2.5,  3.5 ]
 [ -2.5, 2,   -2.5 ]]
```

Integer Matrix Data

```
# Template for Agilent Ptolemy integer matrix data
# Each matrix data set separately listed with brackets around each row and
matrix
# Each matrix row separated by newlines
[[ 1 -2,  2 ]
 [ -2, 2, -2 ]]
[[ 2, -2,  3 ]
 [ -3, 2, -1 ]]
[[ 2, -2,  3 ]
 [ -2, 2, -2 ]]
```

Complex Matrix Data

```
# Template for Agilent Ptolemy complex matrix data
# Each matrix data set separately listed with brackets around each row and
matrix
# Each matrix row separated by newlines
[[ 11.0+0.0j, 12.0+0.0j, 13.0+0.0j ]
 [ 21.0+0.0j, 22.0+0.0j, 23.0+0.0j ]]
```

SPW (.ascsig and .sig) File Formats

SPW format data files can be read by the system simulator by specifying them as input files in a TimeFile component, or as output files from an OutFile component. The binary format *.sig* file has the same ASCII header information as the *.ascsig* file but the data is stored as a pointer in binary format.

1. The SPW version 3.0 data file format must be used.
2. Comments can only be included on the one line following the \$USER_COMMENT statement.
3. The TimeFile source can read a real double-format or complex double-format SPW data file. To read an SPW format file, the appropriate *.ascsig* or *.sig* extension must be specified with the filename.

Real Double Data Format Example .ascsig File

```
$SIGNAL_FILE 9
$USER_COMMENT

$COMMON_INFO
SPW Version = 3.0

Sampling Frequency = 1
Starting Time = 0
$DATA_INFO
Number of points = 6
Signal Type = Double
$DATA
1.00000000000000000000000000000000
1.00000000000000000000000000000000
-1.00000000000000000000000000000000
-1.00000000000000000000000000000000
1.00000000000000000000000000000000
1.00000000000000000000000000000000
END
```

(Note: There is no space between the sign and the number, e.g., -1.0.)

Complex double data format example .ascsig file

```
$SIGNAL_FILE 9
$USER_COMMENT

$COMMON_INFO
```

Understanding File Formats

SPW Version = 3.0

```
Sampling Frequency = 1
Starting Time = 0
$DATA_INFO
Number of points = 10
Signal Type = Double
Complex Format = Real_Imag
$DATA
1.00000000000000000000000000000000+j1.00000000000000000000000000000000
1.00000000000000000000000000000000+j1.00000000000000000000000000000000
-1.00000000000000000000000000000000+j1.00000000000000000000000000000000
-1.00000000000000000000000000000000+j1.00000000000000000000000000000000
1.00000000000000000000000000000000+j1.00000000000000000000000000000000
1.00000000000000000000000000000000+j1.00000000000000000000000000000000
-1.00000000000000000000000000000000+j1.00000000000000000000000000000000
-1.00000000000000000000000000000000+j1.00000000000000000000000000000000
-1.00000000000000000000000000000000+j1.00000000000000000000000000000000
-1.00000000000000000000000000000000+j1.00000000000000000000000000000000
END
```

(Note: There is no space between the sign and the number, e.g., -1.0.)

Time-Domain Waveform Data (.tim) File, MDIF ASCII Format

The general *.tim* file format is:

```
BEGIN TIMEDATA
#   T   ( SEC   V   R   xx )
%   t   v
<data line>
.
.
.
<data line>
END
```

Guidelines for .tim files

An exclamation point (!) at the beginning of a line makes it a comment line; characters following ! are ignored by the program.

TIMEDATA data block is required.

- The BEGIN statement:

```
BEGIN TIMEDATA      ! Begin time-domain waveform data
```

- Option line:

```
# T ( [SEC/MSEC/USEC/NSEC/PSEC] V/MV/DATA R xx )
```

where

= delimiter that tells the program you are specifying these parameters

T = time

SEC/MSEC/USEC/NSEC/PSEC = seconds/milliseconds/microseconds/
nanoseconds/picoseconds

V/MV/DATA = volts/millivolts/data

R = reference resistance, default 50.0

xx = user-specified value for reference resistance

(Note that values R and xx are for documentation purposes and are not used by the simulator.)

- Format line:

```
% t v
```

where

% = delimiter that tells the program you are specifying these parameters

t = time

v = voltage

- TIMEDATA data requirements are:

- A value for t=0 is not required.
- The signal is assumed to be time periodic with time period equal to maximum time minus minimum time.
- The DATA option line value specifies that voltage values are to be interpreted as discrete voltage levels with no interpolation between consecutive voltage points, with each voltage level held constant until the next voltage point.

Figure 6-1 shows two *.tim* example files that result in a time periodic voltage versus time with time period 32 μ sec. Example 1 is interpreted as a piecewise linear voltage description; example 2 is interpreted as a voltage state description.

Time-Domain Waveform Data (.bintim) File in Binary Format

The general *.bintim* file format is:

```
NUMBER OF DATA XX1  
BEGIN TIMEDATA  
# T ( SEC V R XX)  
% T V  
<binary data block>
```

For binary (*.bintim*) the line NUMBER OF DATA is added as the first file line to specify the number of time-voltage pairs, XX1. The begin, option, and format lines follow the same rules as for the *.tim* file. And, there is no *END* line.

Agilent Standard Data Format (.dat) Files

The *.dat* file is a signal file form used with the Agilent 89400 and 89600 series of test instruments (vector modulation generators/analyzers). Refer to the Agilent *Standard Data Format Utilities User's Guide*, Agilent Part No. 5061-8056.

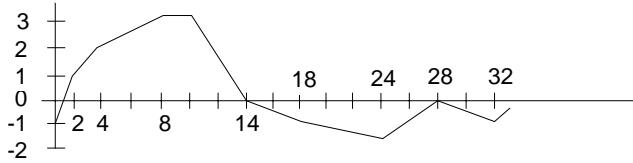
Example 1

```
BEGIN TIMEDATA
# T ( USEC V R 50. )
% t v
0.0 -1.0
2.0 1.0
4.0 2.0
8.0 3.0
10.0 3.0
14.0 0.0
18.0 -1.0
24.0 -2.0
28.0 0.0
32.0 -1.0
END
```

Example 2

```
BEGIN TIMEDATA
# T ( USEC DATA R 50. )
% t v
0.0 -1.0
2.0 1.0
4.0 2.0
8.0 3.0
10.0 3.0
14.0 0.0
18.0 -1.0
24.0 -2.0
28.0 0.0
32.0 -1.0
END
```

Example 1



Example 2

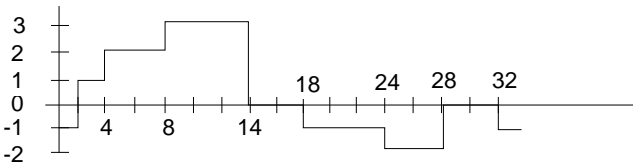


Figure 6-1. Example .tim Files

Chapter 7: Performing Parameter Sweeps

This chapter discusses performing parameter sweeps with Agilent Ptolemy. This capability works in a similar manner to Advance Design System's Analog/RF Systems simulators. In this chapter, we will describe this capability using signal processing examples and point out some things to be aware of when performing sweeps with Agilent Ptolemy.

Introduction

Parameter sweeps are a quick way to conduct a series of simulations while varying a parameter and displaying the output on one plot.

For example, you could analyze a bit error rate (BER) measurement while sweeping the amount of noise added to the design.

Note In many of Advanced Design System's circuit simulators, sweeps of individual parameters (such as frequency) can be performed from within many of the simulator dialog boxes themselves. However, in signal processing, a Parameter Sweep component must be used.

Sweeping Parameters Using the ParamSweep Component

To sweep parameters, you need to place and specify parameters for the ParamSweep component. Alternately, you can place a VAR component (variables and equations) to aid in defining terms.

Both ParamSweep and VAR components are found in the *Controllers* component library. To learn how to use parameter sweeps, we will build a simple design, as shown in [Figure 7-1](#).

Performing Parameter Sweeps

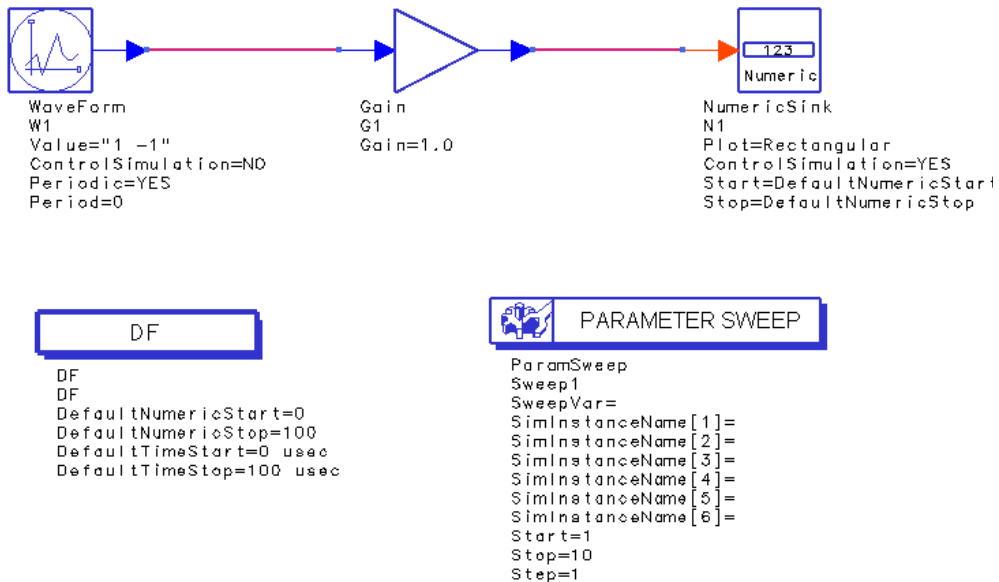
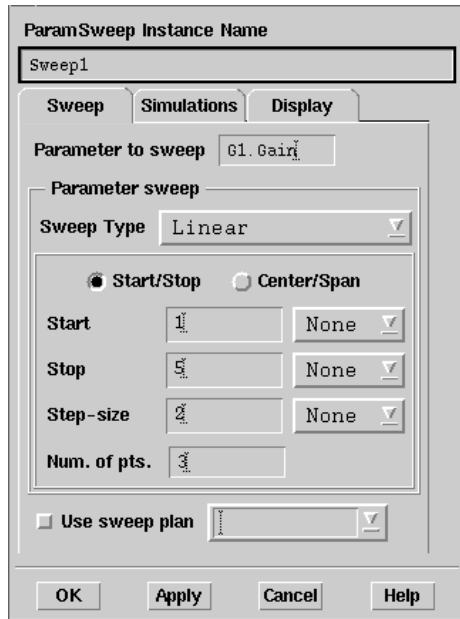


Figure 7-1. Sweeping a Gain Component

In this design, we have placed a Waveform source, a Gain component, and a Numeric Sink (from the Common Components library). Default values are used for each. We have also placed the required Data Flow controller and a ParamSweep component. The parameter we will sweep is the gain of component G1.

Procedure

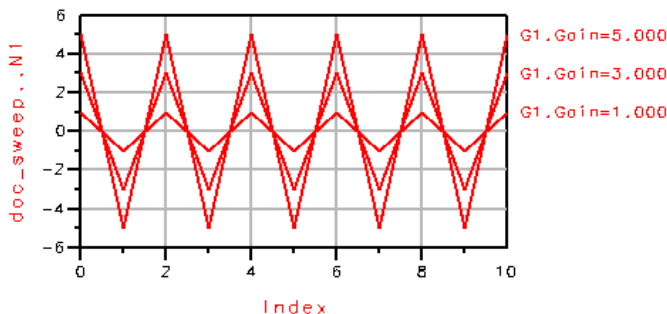
1. Double-click the **ParamSweep** component to display the Component Parameters dialog box, as shown next.



2. With the Sweep tab displayed (default), in the *Parameter to sweep* field (near top), enter **G1.Gain**. G1 is the instance name of the Gain component. A period separates this name from the parameter we want to sweep (Gain). With this syntax, you can set up any parameter of any component for sweeping. Additionally, this syntax is hierarchical. If the parameter Gain was in a subnetwork called A, you would use A.G1.Gain.
3. Leave the Sweep Type field set to **Linear**. Other choices are single point and log.
4. With the Start/Stop option button selected, enter the following parameters:
Start = 1
Stop = 5
Step-size = 2
Num. of pts. = 3 (this field is calculated by the program)
5. Now switch to the **Simulations** tab.
6. In the Simulations to perform field, enter **DF1** next to Simulation 1. If this field is left blank, the simulation will not be successful. Here you are telling the

program to use the Data Flow controller specified by the instance name DF1 as the simulator. In rare occasions, you may have multiple Data Flow controllers, each with its own instance name.

7. Click **OK** to accept your changes and dismiss the dialog box.
8. Double-click the Data Flow controller component to edit its parameters.
9. Change the Stop parameter to **10.0**.
10. Choose **Simulate > Simulate**.
11. When the simulation is finished, choose **Window > New Data Display**.
12. Place a rectangular plot, add N1 from your dataset, and choose **OK**. Your result should indicate the waveform multiplied by three different Gain values and look similar to the one shown in the next figure.



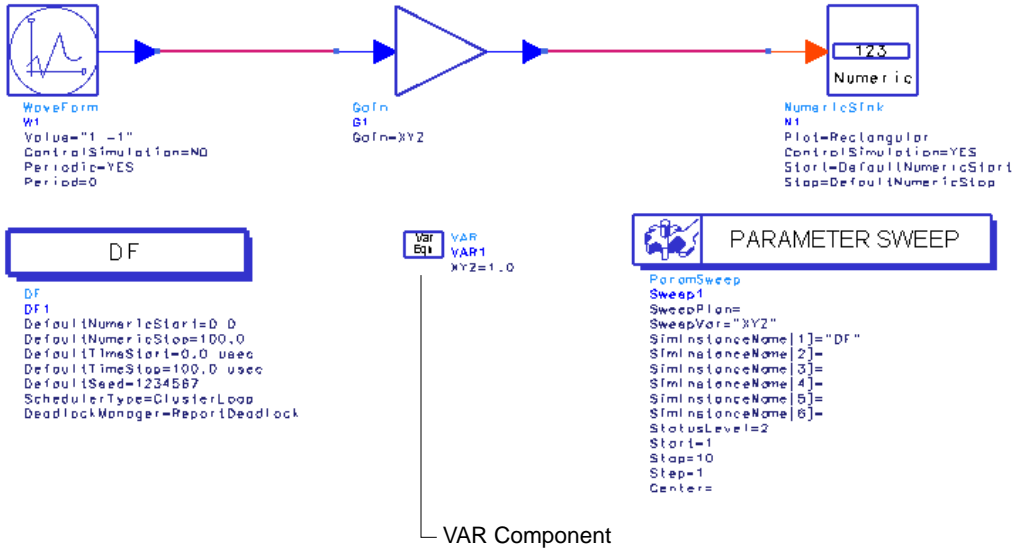
Sweeping Parameters Using the VAR Component

An alternate way to conduct parameter sweeps is to place a VAR component. A VAR component is used to define variables and equations, and you place one in addition to the ParamSweep component.

The following procedure uses the same design as before, but this time we will place VAR component to help perform the parameter sweep.

Procedure

1. Place a VAR component from the Controllers library anywhere in the schematic.



2. Edit the Gain component so that **Gain=XYZ** (instead of the default of Gain=1.0).
3. Edit the VAR component so that **XYZ=1.0**.
4. Edit the ParamSweep component so that **SweepVar="XYZ"** and **SimInstanceName[1]="DF"**. Leave the rest of the ParamSweep parameters unchanged.

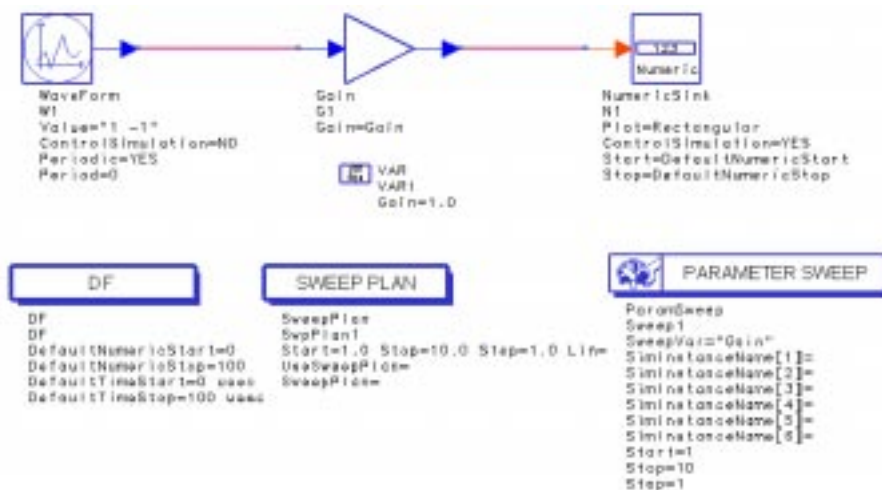
In this method, you are connecting the Gain parameters with the sweep variable XYZ. Any reference to XYZ in this design would be swept. Further, you may want to use VAR components to define other relationships in a design and add another line to define the parameter to be swept. The use of VAR components allows you a flexible way of building complicated sweep relationships. However, for a simple parameter sweep, it is easiest to use only the ParamSweep component.

Using a Sweep Plan

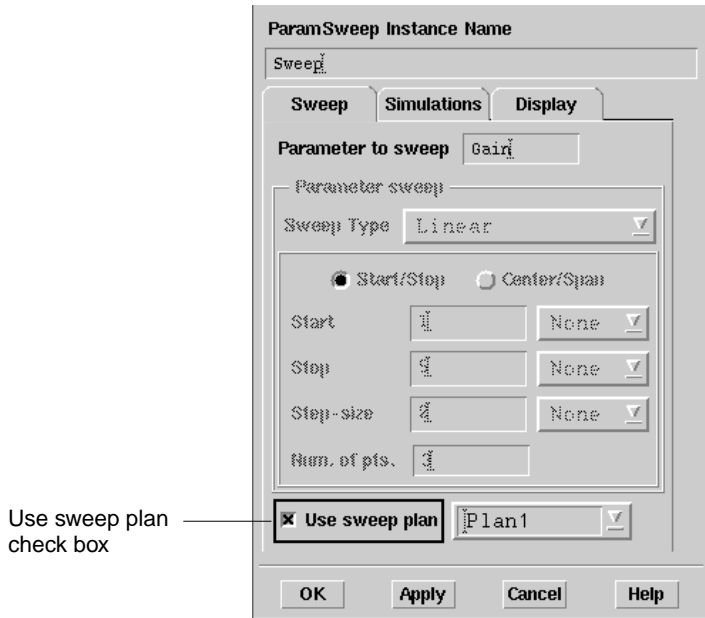
It is possible to combine sweeps of several parameters or several ranges of one parameter into a single sweep plan. This plan of multiple parameter sweeps is controlled by placing a Sweep Plan in your schematic.

Procedure

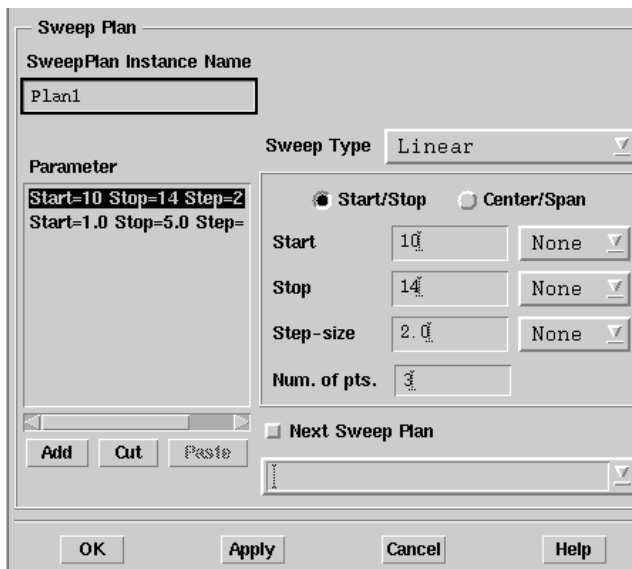
1. Place a Sweep Plan component from the Controllers library anywhere in the schematic.



2. Edit the Parameter Sweep component so that the *Use sweep plan* check box is selected, as shown below. The Sweep tab must be selected before you can check this box.



3. Double-click the Sweep Plan component to edit its parameters.
4. Enter two ranges of gain steps. First, in the Start/Stop field repeat the range that was entered for the single parameter sweep:
 - Start = 1
 - Stop = 5
 - Step-size = 2
5. Choose **Apply**.
6. Next, change the Start/Stop field to:
 - Start = 10
 - Stop = 14
 - Step-size = 2
7. Choose the **Add** button (left side). This adds our new range to the sweep plan, as shown below.

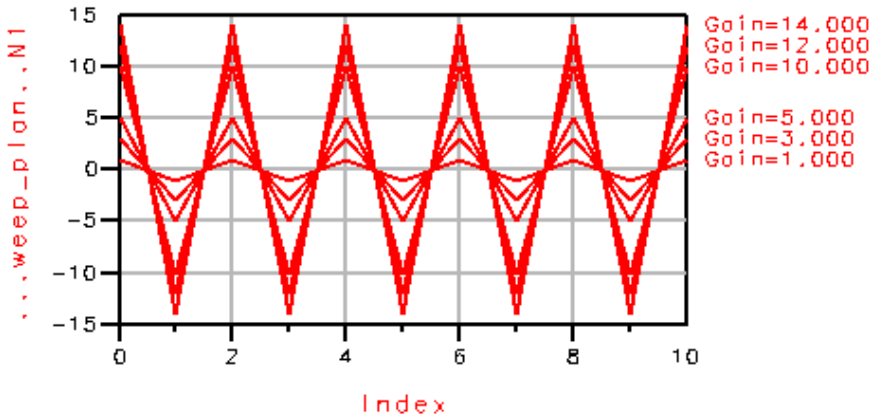


8. Click **OK**.

It is possible to place multiple Sweep Plan components that control simulation in a chain of events. To do this, you place additional Sweep Plan components and choose the *Next Sweep Plan* check box (in the lower-center of the Sweep Plan dialog box).

9. Choose **Simulate > Simulate**.

When the simulation is finished, your Data Display window should look similar to the following:



The Gain parameter has now been swept over two ranges.

Note The placement of sweep components within a design does not affect the order in which parameters are swept. Similarly, the order in which the sweeps are automatically numbered does not determine the order in which they are executed. The order of execution is determined by the order in which one sweep calls another, as determined by the value of the SweepPlan. The simulation component calls the first sweep plan to be conducted, whatever it is named.

Sweeping Various Parameter Types

If you want to sweep a Real, Integer, or Fixed Point parameter type, the procedure is similar to the example just presented. However, if you want to sweep

- Complex
- Precision
- Array
- String
- or Filename

parameter types, you have to perform additional steps.

To sweep these parameter types, you must use a VAR (Variables and equations) component to define the swept variable. You then embed a variable from the VAR in the string of the component parameter value. The reason for this procedure is the simulator only sweeps numbers and these parameter types are strings that are interpreted by the simulator.

An example of sweeping a filename is the case of ten files, myfile1.dat through myfile10.dat each containing filter coefficients. You might want to sweep a range of these files.

Note Earlier versions of Advanced Design System (1.0 and 1.1) required the use of *sprintf* and *strcat* functions to reference strings. While no longer needed for complex, precision, or array types, designs built with these functions will still work.

Sweeping a Complex Waveform Component

This example, shown in [Figure 7-2](#), shows how to sweep a parameter type that is represented by a string.

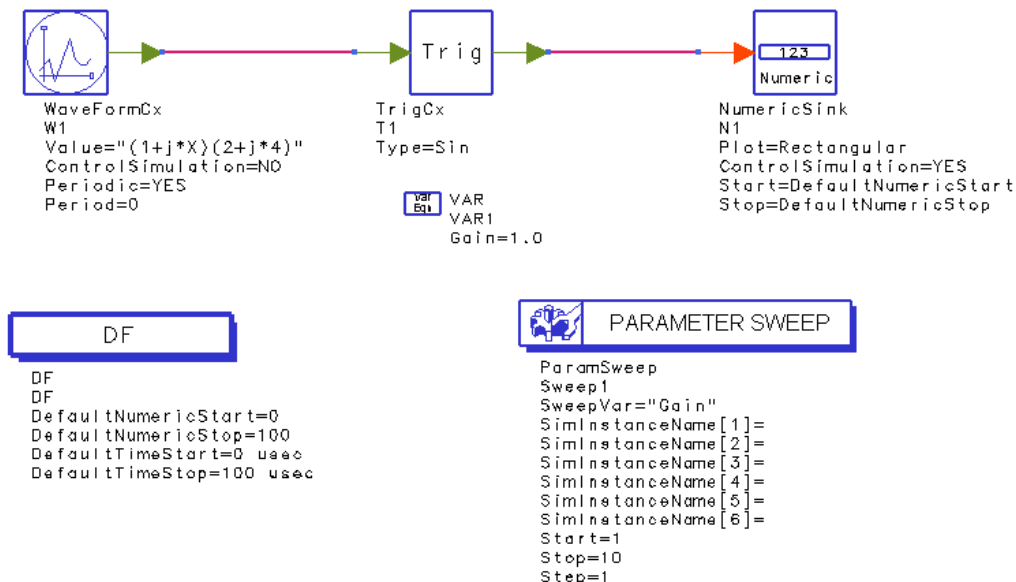
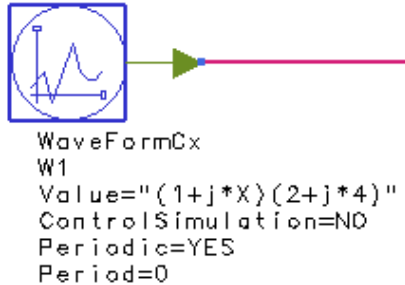


Figure 7-2. Sweeping a Complex Waveform Component's Value

The basic parameter sweep set up is similar to our earlier example. Here we use a VAR (Variables and equations—symbol Var Eqn) component (from the Controllers library or palette) to define the swept variable called X. Let's zoom in on the WaveformCx component, which produces a complex waveform.

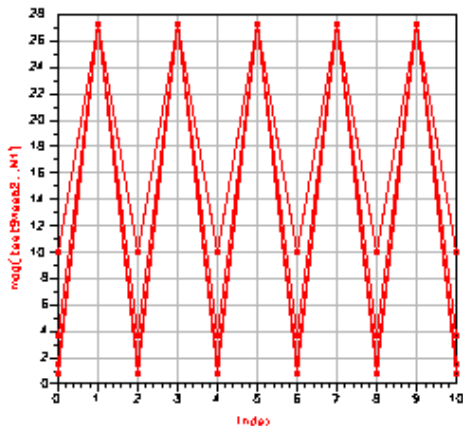


The parameter we want to sweep is the imaginary part of the first complex entry in an array of two complex numbers. Since complex arrays are handled as strings in Agilent Ptolemy, we sweep the imaginary part as follows:

```
Value="(1+j*X)(2+j*4)"
```

The string "(1+j*X)(2+j*4)" tells the software to evaluate the mathematical expression for variable X and convert it to a string.

Now if we sweep the variable X from 0 to 3, we obtain the following result for the magnitude of the output:



Performing Multidimensional Sweeps

Sometimes you need to sweep two or more variables and observe the composite results. An example is analyzing the bit error rate (BER) of a communication system for two modulation schemes at three different power levels. Here, for each of two modulation formats X, there are three power levels Y. You set up this type of simulation using two Parameter Sweep components.

An example of another multidimensional sweep design is shown [Figure 7-3](#):

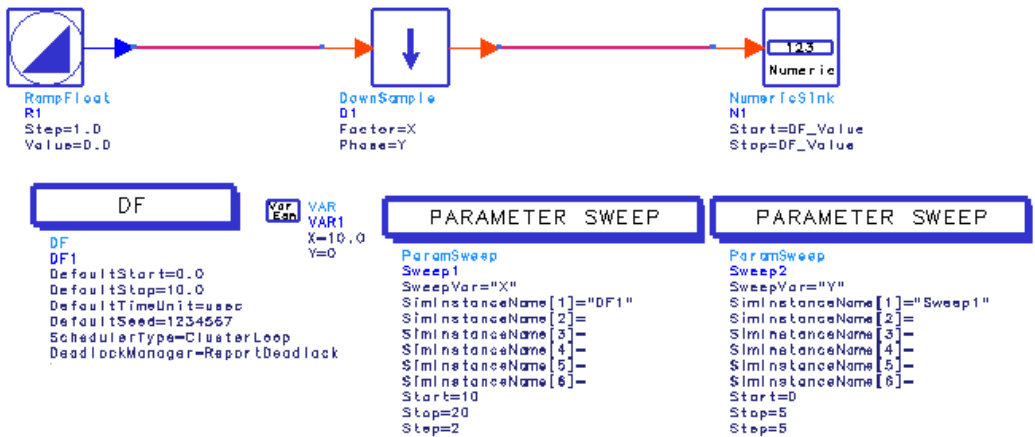


Figure 7-3. Multidimensional Sweep Example Using a DownSample Component

In this example, a RampFloat is used as a source for a DownSample component, with the results stored in a Numeric Sink. The parameters Factor=X and Phase=Y of the DownSample component are swept simultaneously. X is swept from 10 to 20 in steps of 2. Y is swept from 0 to 5 in steps of 5. There are a total of 6*2 traces in the resulting Data Display window, shown [Figure 7-4](#):

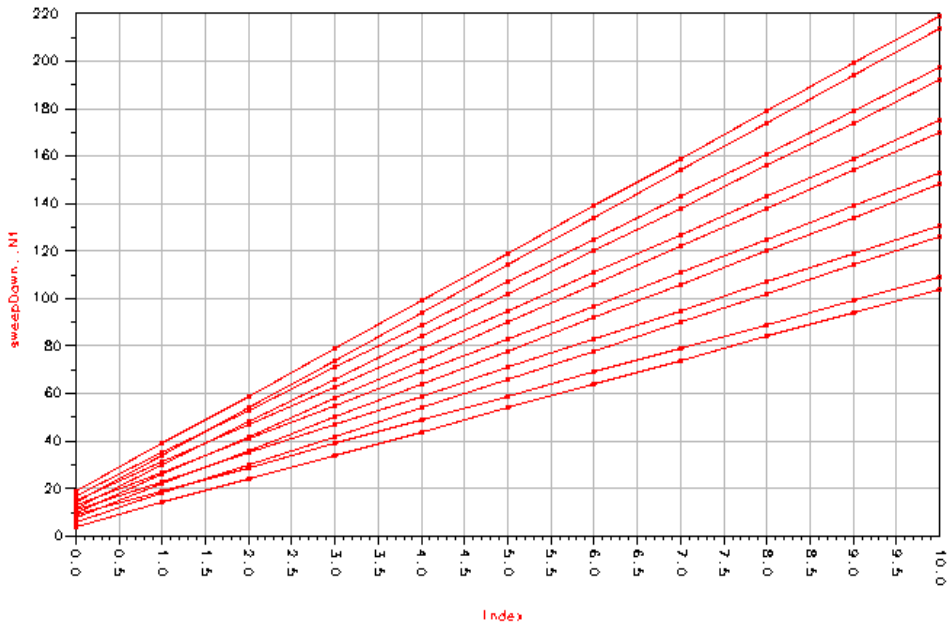


Figure 7-4. Simulation Results of Multidimensional Sweep Example
Each line is associated with a downsampling factor (X) for a given phase (Y).

Chapter 8: Using Nominal Optimization

This chapter describes using Nominal Optimization with Agilent Ptolemy. Nominal optimization, also called performance optimization, employs iterative simulation to achieve user-specified goals by automatically varying specific simulation design parameter values over user-specified ranges. For example, you could optimize the gain of a carrier recovery loop to achieve a desired lock time and residual loop error or you could optimize a fixed-point bit-width parameter in a DSP design

This capability generally works the same way as in Advance Design System's Analog/RF Systems simulators. The Nominal Optimization procedure is found in the manual *Tuning, Optimization, and Statistical Design*. Refer to *Performing Nominal Optimization* that manual for complete details.

In this chapter we will present a DSP example on optimizing bit-width and describe some additional information on signal processing parameter types to be aware of when performing optimization with Agilent Ptolemy.

Optimizing Various Parameter Types

If you want to optimize a Real, Integer, or Fixed Point parameter type, the procedure is similar to standard nominal optimization. However, if you want to optimize

- Complex
- Precision
- Array
- String
- or Filename

parameter types, you have to perform additional steps.

To optimize these parameter types, you must use a VAR component to define the optimizable variable. You then embed a variable from the VAR in the string of the component parameter value. The reason for this procedure is the simulator only sweeps numbers and these parameter types are strings that are interpreted by the simulator.

To reference an optimized variable for parameter types that use strings, the variable is defined in a VAR (Variables and equations—symbol Var Eqn) component with the Standard entry mode and Optimization/Statistic Setup enabled. Once defined, this

variable may be used as a component parameter, as shown in Example 2, below. If you type this variable on the schematic, it must be enclosed in quotes (“ ”). If you enter it in the Component dialog box, quotes are automatically added.

An example of optimizing a filename is the case of ten files, myfile1.dat through myfile10.dat, each containing filter coefficients. You might want to conduct an optimization based on a range of these files.

Note Earlier versions of Advanced Design System (1.0 and 1.1) required the use of *sprintf* and *strcat* functions to reference strings. While no longer needed for complex, precision, or array types, designs built with these functions will still work.

Optimizing Input and Output Bit Width

This example shows how to set up a simple fixed-point bit-width parameter optimization. To learn how to do this optimization, we will build a simple design, as shown in [Figure 8-1](#). You can copy it from the *examples/Tutorial* directory. The project file is *dspopt_prj* and the design is *simpleopt2*. Or, you can quickly build this example or just follow along from this manual.

- A Data Flow controller, with default values accepted, except set Default Stop to **0**.
- A VAR component, set up as described in the next section.
- An Optim component, set up as described in the section *Setting Optimization Job Parameters* in the *Tuning, Optimization, and Statistical Design* manual. Use the default values, except set MaxIters = **30**, and P = **2**. The default optimizer type is the **Random** optimizer.
- A Goal component, for the expression **N1**.
- A second Goal component, for the expression **N2**, set up as described in the section [“Setting Up the ConstInt, Second Sink, and Second Goal Components” on page 8-5](#).

The value of 0.2 cannot be exactly represented with only one or two fractional bits (bits to the right of the decimal point). Without optimization, too many bits (such as 16) might be used. While the number 0.2 would be represented very accurately, the extra bits could be wasteful in the final implementation.

By studying this simple example, you can learn the procedure you would use to solve real-world design problems, such as optimizing the bit width of an FIR filter.

Setting Up the VAR Component

Let’s zoom in on the VAR component:

```

Var  VAR
Eqn  VAR2
     W=2
     D=1 opt{ discrete 1 to 16 by 1 }

```

Figure 8-2. VAR Component Parameters

Double-click the VAR component in your schematic to display its dialog box. The following is a description of each parameter we have to set so that the VAR component matches [Figure 8-2](#):

Note The labels, such as W or D, are user-defined variable names and could be anything you wanted.

1. Enter **W= 2**. W is the number of bits to the left of the decimal point, including the sign bit.
2. Enter **D=1**. D is the number of bits to the right of the decimal point. 1 is our nominal value before optimization. This should be your best estimate for the nominal value.
3. After you enter D=1, choose the **Optimization/Statistics Setup** button.
4. From the Optimization Status drop-down list, select **Enabled**.
5. From the Type drop-down list, select **Discrete**.
6. In the Minimum Value field, enter **1**.
7. In the Maximum Value field, enter **16**.
8. In the Step Value field, enter **1**.

These last four steps tell the program to optimize using only the discrete values of 1 through 16 in steps of 1.

9. Click **OK** to return to the main Variables and Equations dialog box.
10. When done, click **OK**.

Setting Up the ConstInt, Second Sink, and Second Goal Components

Optimizing bit width requires placing a second source, ConstInt; a second Numeric Sink; and a second Goal component. Wire the second ConstInt to the second Numeric Sink. The purpose of these secondary components is to create a second optimization goal, which the program will attempt while meeting the criteria of the first optimization goal. Do the following:

1. Edit the ConstInt Component to set **Level=D**. D is the number of bits to the right of the decimal point, that you defined in the VAR component.
2. Accept the defaults for the second Numeric Sink, N2.
3. Edit the parameters for the second Goal component, OptimGoal2, as follows:

```
Expr = "N2"  
SimInstanceName = "DF1"  
Min = 0  
Max = 0
```

Weight = 1
RangeVar = Index
RangeMin = 0
RangeMax = 0

Weight Parameter

Keep in mind that the Weight parameter weighs the importance of one goal to the other goal(s). Generally, the first goal may be more important, such as when it meets a performance specification, such as frequency response. The second goal (in this example, bit width) is weighted less. Because the error function of the first goal is small compared to the second goal, the Weight of the first goal is set to 1 e9.

Completing the Optimization

You are now ready to complete the optimization.

The remainder of the procedure for completing and running the optimization for parameter types such as precision or string, as described in this section, are the same as for any optimization. To review these procedures, refer to *Specifying Component Parameters for Optimization* in the *Tuning, Optimization, and Statistical Design* manual for details.

Chapter 9: Theory of Operation

Introduction

Agilent Ptolemy provides signal processing simulation for Advanced Design System's specialized design environments. Each of these specialized design environments capture a model of computation, called a domain, that has been optimized to simulate a subset of the communication signal path. Advanced Design System domains that are part of Agilent Ptolemy, or can cosimulate with Agilent Ptolemy are:

Domain	Simulation Technology	Controller	Application Area
Synchronous Dataflow (SDF)	Numeric dataflow	Data Flow	Synchronous multirate signal processing simulation
Timed Synchronous Dataflow (TSDF)	Timed dataflow	Data Flow	Baseband and RF functional simulation (e.g., antenna and propagation models, timed sources)
Circuit Envelope	Time- and frequency-domain analog	Envelope	Complex RF simulation
Transient	Time-domain analog	Transient	Baseband analog simulation

In Agilent Ptolemy, a complex system is specified as a hierarchical composition (nested tree structure) of simpler circuits. Each subnetwork is modeled by a domain. A subnetwork can internally use a different domain than that of its parent. In mixing domains, the key is to ensure that at the interface, the child subnetwork obeys the semantics of the parent domain.

Thus, the key concept in Agilent Ptolemy is to mix models of computation, implementation languages, and design styles, rather than trying to develop one, all-encompassing technique. The rationale is that specialized design techniques are more useful to the system-level designer, and more amenable to a high-quality, high-level synthesis of hardware and software. In the next sections we will describe the SDF and TSDF. For general documentation on the Circuit Envelope and Transient simulators refer to the Circuit Envelope and RF Transient/Convolution Simulation chapters in the *Circuit Simulation* manual. For information on cosimulation with Agilent Ptolemy and these circuit simulators, refer to [Chapter 11, Cosimulation with Analog/RF Systems](#).

Synchronous Dataflow (SDF)

SDF is a special case of the dataflow model of computation, which was developed by Dennis¹. The specialization of the model of computation is to those dataflow graphs where the flow of control is completely predictable at compile time. It is a good match for synchronous signal processing systems, those with sample rates that are rational multiples of one another.

The SDF domain is suitable for fixed and adaptive digital filtering, in the time or frequency domains. It naturally supports multirate applications, and its rich component library includes polyphase FIR filters.

Advanced Design System's Examples directories contain numerous application examples which rely on SDF semantics. To view these examples, chose *File > Example Project*, a dialog box appears. Select the *DSP/dsp_demos_prj* directory for one group of SDF examples.

Synchronous dataflow (SDF) is a data-driven, statically scheduled domain in Agilent Ptolemy. It is a direct implementation of the techniques given by Lee^{2,3}. *Data-driven* means that the availability of data at the inputs of a component enables it. Components without any inputs are always enabled. *Statically scheduled* means that the firing order of the components is periodic and determined once, during the start-up phase. It is a simulation domain, but the model of computation is the same as that used for bit-true simulation of synthesizable hardware used by the DSP Synthesis tool. A number of different schedulers have been developed for this model of computation.

Basic Dataflow Terminology

The SDF dataflow model is equivalent to the *computation graph* model of Karp and Miller⁴. In the terminology of the dataflow literature, components are called *actors*. An invocation of a component is called a *firing*. The signal carried along the arc connecting the blocks are made of individual packets of data called *tokens*. In a digital signal processing system, a sequence of tokens might represent a sequence of samples of a speech signal or a sequence of frames in a video sequence.

Note Some Agilent Ptolemy terminology is different from UCB Ptolemy terminology. For example, a *component* in Advanced Design System is called a *star* in UCB Ptolemy and an *arc* is a *wire*. Refer to the Glossary for more information.

When an actor fires, it consumes some number of tokens from its input arcs, and produces some number of output tokens. In synchronous dataflow, these numbers remain constant throughout the execution of the system. It is for this reason that this model of computation is suitable for synchronous signal processing systems, but not for asynchronous systems. The fact that the firing pattern is determined statically is both a strength and a weakness of this domain. It means that long runs can be very efficient, a fact that is heavily exploited in Agilent Ptolemy and the Agilent DSP Synthesis tool. But it also means that data-dependent flow of control is not allowed. This would require dynamically changing firing patterns.

Balancing Production and Consumption of Tokens

Each port of each SDF component has an attribute that specifies the number of tokens consumed (for inputs) or the number of tokens produced (for outputs). When you connect an output to an input with an arc, the number of tokens produced on the arc by the source component may not be the same as the number of tokens consumed from that arc by the destination component. To maintain a balanced system, the scheduler must fire the source and destination components with different frequency.

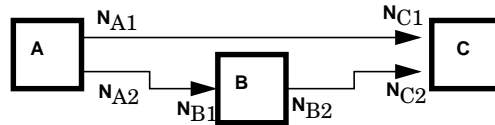


Figure 9-1. Simple Connection of SDF Components Illustrates Balance Equations Constructing a Schedule

Consider a simple connection between three components, as shown in [Figure 9-1](#). The symbols adjacent to the ports, such as N_{A1} , represent the number of tokens consumed or produced by that port when the component fires. For many signal processing components, these numbers are simply one, indicating that only a single token is consumed or produced when the component fires. But there are three basic circumstances in which these numbers differ from one:

- Vector processing in the SDF domain can be accomplished by consuming and producing multiple tokens on a single firing. For example, a component that computes a fast Fourier transform (FFT) will typically consume and produce 2^M samples when it fires, where M is some integer. Examples of vector processing components that work this way are FFT_Cx, Average, and FIR. This behavior is

quite different from the matrix components, which operate on tokens where each individual token represents a matrix.

- In multirate signal processing systems, a component may consume M samples and produce N , thus achieving a sampling rate conversion of N/M . For example, the FIR component can optionally perform such a sampling rate conversion, and with an appropriate choice of filter coefficients, can interpolate between samples. Other components that perform sample rate conversion include UpSample, DownSample, and Chop.
- Multiple signals can be merged using components such as Commutator or a single signal can be split into subsignals at a lower sample rate using the Distributor component.

To be able to handle these circumstances, the scheduler first associates a simple balance equation with each connection in the graph. For the graph in [Figure 9-1](#), the balance equations are

$$r_A N_{A1} = r_C N_{C1}$$

$$r_A N_{A2} = r_B N_{B1}$$

$$r_B N_{B2} = r_C N_{C2}$$

This is a set of three simultaneous equations in three unknowns. The unknowns, r_A , r_B , and r_C are the *repetitions* of each actor that are required to maintain balance on each arc. The first task of the scheduler is to find the smallest non-zero integer solution for these repetitions. It is proven in Lee¹ that such a solution exists and is unique for every SDF graph that is “consistent,” as defined below.

How Schedulers Work in Agilent Ptolemy

Synchronous Dataflow (SDF) is a restricted version of dataflow in which the number of data produced or consumed by a component per invocation is known at compile time. As the name implies, SDF can be used to model synchronous signal processing algorithms. In these algorithms, all of the sampling rates are rationally related to one another.

An SDF scheduler takes a simulation design and determines the sequence or order of invocation of each component. The simulator will simulate the design according to the schedule generated by the scheduler. Agilent Ptolemy provides three user-selectable schedulers for any given simulation:

- Classical Scheduler
- Cluster Loop Scheduler
- Acyclic Loop Scheduler

The trade-offs between them are typically the time needed to generate the schedule, the memory usage of storing the schedule data structure, and the memory usage of buffers for data collection.

Classical Scheduler

This is the classic scheduler from UC Berkeley Ptolemy, which tries to minimize the buffer sizes. It tries to put off the firing of any block as long as possible. A component firing is deferred until none of the components that feeds data to it can be fired. It usually takes longer to generate the schedule than other schedulers, and has a large schedule size, but it uses less memory buffers across components. This scheduler is good for uni-rate designs.

Cluster Loop Scheduler

This scheduler generates single-appearance schedules that take less time than the Classical scheduler. The advantage of a single-appearance schedule is that it significantly reduces schedule size. Each component appears once in the schedule and possibly with a loop factor. However, the buffer memory usage will be increased by the loop factor. Most of the increased buffer memory usage can be reduced by clustering components and limiting the buffer increases to only between clusters. This scheduler is good for multirate designs such as wireless 2.5/3G designs.

Acyclic Loop Scheduler

This scheduler generates single-appearance looped schedules for acyclic graphs to optimizing both the schedule and buffer sizes by using a “recursive partitioning by minimum cuts” heuristic. More information about the algorithms and heuristics can be found in chapters 6 and 7 of *Software Synthesis* from Dataflow Graphs (<http://ptolemy.eecs.berkeley.edu/~murthy/book.html>) by Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee, published by Kluwer Academic Publishers, Norwood, MA, 1996. This scheduler is good for multirate designs without cycles.

Iterations in SDF

At each SDF iteration, each component is fired the minimum number of times to satisfy the balance equations.

Suppose for example that component B in [Figure 9-1](#) is an FFT_Cx component with its parameters set so that it will consume 128 samples and produce 128 samples. Suppose further, that component A produces exactly one sample on each output, and component C consumes one sample from each input. In summary,

$$V_{A1} = N_{A2} = N_{C1} = N_{C2} = 1$$

$$N_{B1} = N_{B2} = 128.$$

The balance equations become

$$r_A = r_C$$

$$r_A = 128r_B$$

$$128r_B = r_C.$$

The smallest integer solution is

$$r_A = r_C = 128$$

$$r_B = 1.$$

Hence, each iteration of the system includes one firing of the FFT_Cx component and 128 firings each of components A and B.

Inconsistency

It is not always possible to solve the balance equations. Suppose that in [Figure 9-1](#) we have

$$V_{A1} = N_{A2} = N_{C1} = N_{C2} = N_{B1} = 1$$

$$N_{B2} = 2.$$

In this case, the balance equations have no non-zero solution. The problem with this system is that there is no sequence of firings that can be repeated indefinitely with bounded memory. If we fire A,B,C in sequence, a single token will be left over on the arc between B and C. If we repeat this sequence, two tokens will be left over. Such a

system is said to be *inconsistent*, and is flagged as an error. The SDF scheduler will refuse to run it.

Deadlocks

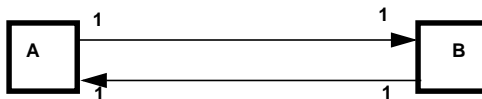


Figure 9-2. A Deadlocked SDF System

While scheduling a system, it is possible that we are not able to complete all of the firings specified in the repetition vector because none of the remaining components are enabled. Such a system is said to be a deadlock state. [Figure 9-2](#) shows a system in such a state. The repetitions vector for this system is:

$$r_A = 1$$

$$r_B = 1.$$

Thus this system is consistent; however, neither A nor B are enabled because each is waiting for one token from the other. The way to resolve this deadlock is to introduce an initial token on one of the two arcs in the figure. This initial token is known as a delay.

The Delay component is indicated by a component with a diamond that is connected to an arc. The delay has a single parameter, the number of samples of delay to be introduced. In the SDF domain, a delay with parameter equal to one is simply an initial token on an arc. This initial token may enable a component, assuming that the destination component for the delay arc requires one token in order to fire. To avoid deadlock, all feedback loops must have delays. The SDF scheduler will flag an error if it finds a loop with no delays. For most token types, the initial value of a delay will be zero.

By default, a delay has a zero value. To specify a specific value for the initial token, use the `InitDelay` token.

There are a number of specialized components in Agilent Ptolemy that add a delay on an arc, such as `DelayRF`, `VcDelayRF`, `ShiftRegPPSyn`, `ShiftRegPSSyn`, `ShiftRegSPSyn`, and `CounterSyn`. `Delay_M` is used for matrices.

Deadlock Resolution

In Agilent Ptolemy, there is an optional deadlock resolution algorithm that can determine where Delay components need to be inserted. If the user desires, the delay components can be automatically spliced in.

For the Data Flow controller, an option item called Deadlock Management is listed. There are three options:

- Report deadlock. The Agilent Ptolemy simulator will simply report deadlocks if the schedule has failed because of a deadlock.
- Identify deadlocked loops. A new algorithm is turned on to locate where the problem occurs. By using this algorithm, Agilent Ptolemy highlights each loop that has deadlocked.
- Resolve deadlock by inserting tokens. Agilent Ptolemy will resolve the deadlock automatically by inserting delays. This option must be used with care. In general, there are many places Agilent Ptolemy can insert a delay to break a deadlock. Each of these cases can lead to different simulation results.

Timed Synchronous Dataflow (TSDF)

TSDF is an extension of SDF described in the previous section. TSDF adds a Timed data type, which is described in [Chapter 5, Using Data Types](#). For each token of the Timed type, both a time step and a carrier frequency must be resolved.

Advanced Design System's Examples directories contain numerous application examples that rely on TSDF semantics. To view these examples, chose *File > Example Project*, a dialog box appears. Select the *DSP/ModemTimed_prj* directory for one group of TSDF examples.

Time Step Resolution

In TSDF, each Timed arc has an associated time step. This time step specifies the time between each sample. Thus the sampling frequency for the envelope of a Timed arc is $1/\text{time step}$.

The sampling frequency is propagated over the entire graph, including both Timed and numeric arcs. To calculate a time step, the SDF input and output numbers of tokens consumed/produced are used.

For any given SDF or TSDF component, the sampling frequency of the component is defined as the sampling frequency on any input (or output) divided by the consumption (or production) SDF parameter on that port. After a sampling frequency is derived for a given component, it is propagated to every port by multiplying the component's rate with the SDF parameter of the port. A sample rate inconsistency error message is returned if inconsistent sample rates are derived.

Carrier Frequency Resolution

Each Timed arc in a timed dataflow system has an associated carrier frequency (F_c). These F_c values are used when a conversion occurs between Timed and other data types, as well as by the Timed components.

The F_c has either a numerical value, which is greater than or equal to zero ($F_c \geq 0.0$), or is undefined ($F_c = \text{UNDEFINED}$). All Timed ports have an associated $F_c \geq 0.0$. Non-timed ports have an UNDEFINED F_c .

During the simulation, all the F_c values associated with all of Timed ports are resolved by the simulator. The resolution algorithm begins by propagating the F_c specified by the user in the Timed sources parameter *Fcarrier* until all ports have their associated F_c . At times, the user may have specified incompatible carrier frequencies, and Agilent Ptolemy will return an error message.

In the feedforward designs, the algorithm will converge quickly to a unique solution. In the designs with feedback, the algorithm takes additional steps to resolve the carrier frequency at all pins.

For feedback paths, a default F_c is assigned by the simulator. This default F_c is then propagated until the F_c converges on the feedback path. This F_c is occasionally non-unique. To specify a unique value, use the SetFc Timed component.

Input/Output Resistance

Resistors can be used with timed components. Resistors provide a means to support analog/RF component signal processing. They provide definition of analog/RF input and output resistance, additive resistive Gaussian thermal (Johnson) noise, and power-level definition for time-domain signals.

Though resistors are circuit components, they are used in the data flow graph by defining their inputs from the outputs of connected TSDF components and their outputs at connected TSDF component inputs.

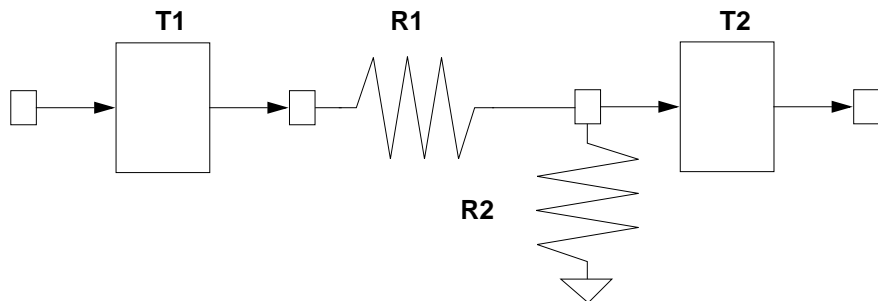


Figure 9-3. TSDF Components with Interconnected Output (R1) and Input (R2) Resistors

The above figure shows two TSDF components, T1 and T2, with an interconnected series resistor, R1, at the output of T1 and a shunt resistor, R2, at the input of T2.

Such interconnected resistors are collected and replaced with the appropriate signal transformation model that includes time-domain signal scaling and additive thermal noise.

Resistors contribute additive thermal noise (kTB) to signals when the specified resistance temperature ($RTemp$) is greater than absolute zero (-273.16 degrees Celsius) where:

k = Boltzmann's constant

T = temperature in Kelvin

B = simulation frequency bandwidth

$1/2 / TStep$: if signal is a timed baseband signal

$1 / TStep$: if signal is a timed complex envelope signal

References

1. J. B. Dennis, *First Version Data Flow Procedure Language*, Technical Memo MAC TM61, May 1975, MIT Laboratory for Computer Science.

2. E.A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24-35, January 1987.
3. E.A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proc. of the IEEE*, vol. 75, no. 9, pp. 1235-1245, September 1987.
4. R.M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing," *SIAM Journal*, vol. 14, pp. 1390-1411, November 1966.

Chapter 10: Introduction to MATLAB Cosimulation

The MATLAB models provide an interface between Agilent Ptolemy and MATLAB, a numeric computation and visualization environment from The MathWorks, Inc. Each MATLAB model can contain a MATLAB function, command, statement, or several statements. Agilent Ptolemy handles the conversion of data to and from MATLAB.

Setting Up MATLAB

Agilent Ptolemy requires version 5 of MATLAB on all platforms.

Under all platforms, set the MATLAB configuration variable in `hpads.cfg` to the root of your MATLAB installation. For example, you might set it to

```
MATLAB=/usr/local/matlab
```

on a UNIX platform, or

```
MATLAB=c:/matlab
```

on a Windows platform. (See the Advanced Design System *Installation Guide* for more details on the `hpads.cfg` file.)

Under UNIX, if the command to invoke MATLAB is not `matlab`, set the `MATLABCMD` configuration variable to the correct command. For example, you might set it to

```
MATLABCMD="matlab -c /path/to/license/file"
```

so that `matlab` correctly finds its license file. Most people won't need to set the `MATLABCMD` variable. The variable's setting is ignored under Windows.

Finally, under Windows, you will need to install MATLAB ActiveX in the Windows registry. To install the entries, run:

```
matlab /Regserver
```

from the command line. MATLAB will register itself and remain running and minimized. At that point, you should exit MATLAB. You need only do this once. For more details, refer to "ActiveX Automation for Windows" in the "Using the MATLAB Engine" chapter of the *MATLAB Application Program Interface Guide*.

If a MATLAB model is run and MATLAB has not been set up correctly, then Agilent Ptolemy will report an error. All MATLAB models in a simulation send their commands to the same MATLAB process.

Simulating with MATLAB

MATLAB distinguishes between real matrices and floating-point matrices. Agilent Ptolemy distinguishes between models that produce data and models that don't. As a consequence, there are three MATLAB models in Agilent Ptolemy, one producing floating-point matrices, one producing complex-valued matrices, and one, a sink, producing nothing. All the models can accept any number of inputs provided that the inputs have the same data type, floating point or complex.

The models are:

Model	Description
Matlab_M	Evaluates a MATLAB expression and outputs the result as floating-point matrices.
MatlabCx_M	Evaluates a MATLAB expression and outputs the result as complex-valued matrices.
MatlabSink	Evaluates a MATLAB expression a fixed number of times.

The models all use a common MATLAB engine interface that is managed by a base MATLAB model. The base model does not have any inputs or outputs. It provides methods for starting and killing a MATLAB process, evaluating MATLAB commands, managing MATLAB figures, changing directories in MATLAB, and passing Agilent Ptolemy matrices into and out of MATLAB. Currently, the base model supports only 2-D real and complex-valued matrices.

The MATLAB interpreter's working directory is set to the ScriptDirectory parameter, if it is given. Any custom MATLAB models will be searched for there, and any output files will be written there.

Figures generated by a MATLAB model are managed according to the value of the DeleteOldFigures parameter. If this parameter is YES, the MATLAB model will close any MATLAB plots or graphics when it is destroyed. If NO, the figures must be manually closed.

Writing Functions for the MATLAB Models

There are several ways in which MATLAB commands can be specified in the MATLAB models in the MatlabFunction parameter.

If only a MATLAB function name is given for this parameter, the function is applied to the inputs in order. The function's outputs are sent to the model's outputs.

For example, specifying *eig* means to perform the eigendecomposition of the input. The function will be called to produce one or two outputs, according to how many output ports there are. If there is a mismatch in the number of inputs and outputs between the Agilent Ptolemy model and the MATLAB function, then an error will be reported by MATLAB.

You may also explicitly specify how the inputs are to be passed to a MATLAB function and how the outputs are taken from the MATLAB function. For example, consider a two-input, two-output MATLAB model to perform a generalized eigendecomposition. The command

```
[output#2, output#1] = eig( input#2, input#1 )
```

says to perform the generalized eigendecomposition on the two-input matrices, place the generalized eigenvectors on output#2, and the eigenvalues (as a diagonal matrix) on output#1. Before this command is sent to MATLAB, the pound characters “#” are replaced with the underscore character “_” because the pound character is illegal in a MATLAB variable name.

The MATLAB models also allow a sequence of commands to be evaluated. Continuing with the previous example, we can plot the eigenvalues on a graph after taking the generalized eigendecomposition:

```
[output#2, output#1] = eig( input#2, input#1 ); plot( output#1 )
```

When entering such a collection of commands in Agilent Ptolemy, both commands appear on the same line without a new line after the semicolon. In this way, very complicated MATLAB commands can be built up. We can make the plot of eigenvalues always appear in the same plot without interfering with other plots generated by other MATLAB models with this function. (New lines are inserted after the semicolons to improve readability.):

```
[output#2, output#1] = eig( input#2, input#1 );  
if ( exist('myEigFig') == 0 ) myEigFig = figure; end;  
figure(myEigFig);  
plot( output#1);
```

The parameters `MatlabSetUp` and `MatlabWriteUp` are called during the model's begin and wrap-up procedures. During each of these procedures, there is no data passing into or out of the model.

Because the same MATLAB interpreter is used for the entire simulation, variables are preserved from iteration to iteration. For example, the output of a `Matlab_M` model with settings:

```
MatlabSetUp = "x=ones(2;1)"  
MatlabFunction = "output#1=x(2)/x(1); x=[x(2),sum(x)];"
```

will converge on the golden mean. It is impossible, however, to share variables between different MATLAB components. Such a simulation would be non-deterministic.

Examples

We recommend studying the MATLAB examples in the Advanced Design System examples directories. From the `./examples/DSP` directory, choose the `dsp_demos_prj` project. The Sombrero network demonstrates how to use each of the three MATLAB models effectively. Sombrero is a simple example that plots a sinc function.

To see an example of how to call a MATLAB `.m` file, refer to the `./examples/DSP/MATLABlink_prj` project.

Chapter 11: Cosimulation with Analog/RF Systems

Simulation of behavioral DSP designs along with analog/RF circuit designs is critical to the success of the integrated components, devices, and subsystems used in today's wireless applications. The need to verify the impact of real-world analog/RF issues on the DSP algorithms and vice versa in a tightly integrated environment is highly desirable.

For designs of low complexity, it is possible to use separate simulators for the signal processing and analog/RF portions and then integrate the results. However, today's state-of-the-art designs using a mix of analog/RF and dedicated on-chip DSP blocks require high levels of integration at the two-environment boundary. Advanced Design System cosimulation between signal processing and circuits addresses this need. Agilent Ptolemy provides the signal processing simulation, while the analog/RF simulation is provided by either the Circuit Envelope or High-Frequency SPICE (Transient) simulators.

Other types of cosimulation include placing MATLAB components or HDL blocks in a signal processing simulation. This chapter describes cosimulation with Analog/RF Systems.

Figure 11-1 shows a mixture of RF circuitry and DSP components. The Advanced Design System provides a variety of analog/RF circuit simulators, including Linear, Harmonic Balance, Circuit Envelope, High-Frequency SPICE, and Convolution. For signal processing simulation, Agilent Ptolemy is used. Only circuits simulated with either Circuit Envelope or High-Frequency SPICE can be instantiated as a subnetwork and included in a signal processing schematic. These circuit blocks can then be simulated along with signal processing components. The steps needed for cosimulation are outlined next.

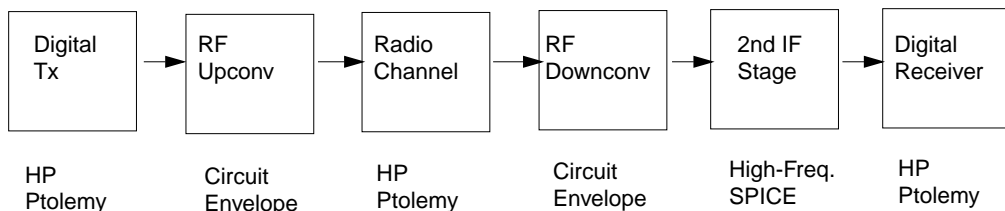


Figure 11-1. Cosimulation: Different Design Portions Simulated by Different Simulators in the Same Schematic

Note Circuit Envelope and High-Frequency SPICE simulators are included with some, but not all, Advanced Design System suites.

Setting Up the Analog/RF Circuit Schematic

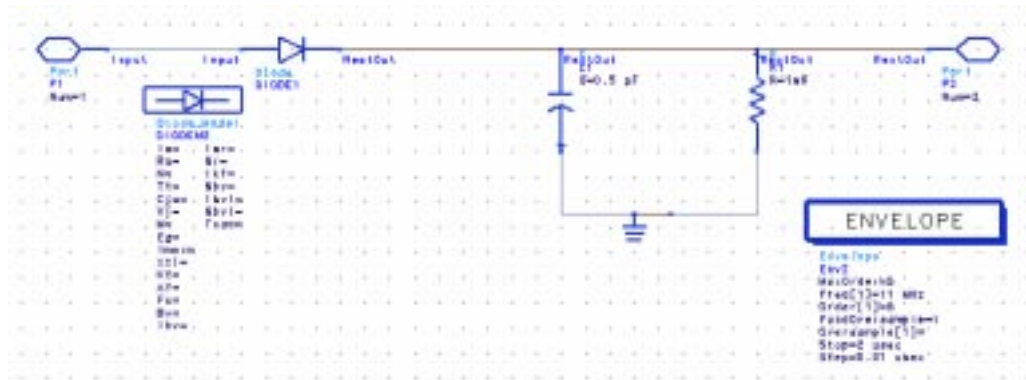


Figure 11-2. Diode Rectifier—Circuit Design Used in Cosimulation

The following general steps are used to create circuit designs for cosimulation.

1. In the analog/RF circuit Schematic window, create a circuit schematic that includes a simulation component for either Circuit Envelope (called ENV) or High-Frequency SPICE simulation (called TRAN).
2. Generally, use Circuit Envelope for an RF simulation and High-Frequency SPICE (transient) for a baseband simulation.
3. Do *not* use both Envelope and Transient simulators in the same design. Use the deactivate and activate commands if you want to keep both controllers in your design.
4. Add ports to your design.
5. Save your design.

In the above figure, a diode rectifier is set to be simulated with the Circuit Envelope simulator. Next, we will place this subnetwork in the signal processing schematic, where it will be represented as a block.

Setting Up the Signal Processing Schematic

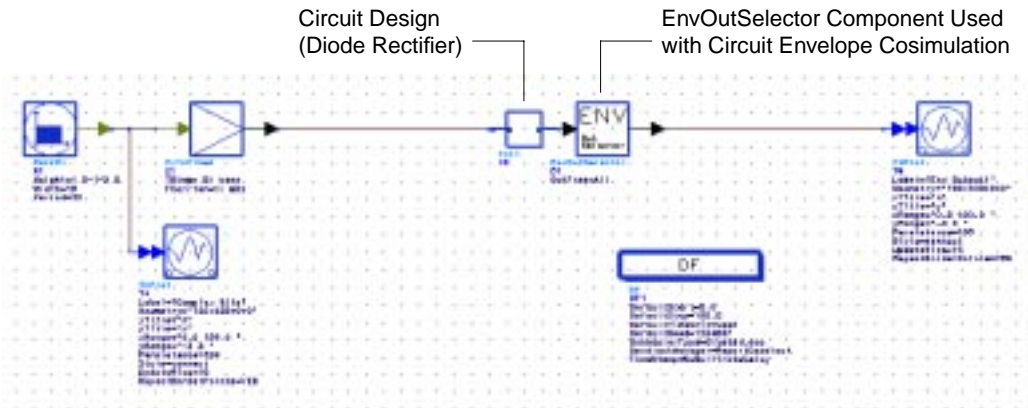


Figure 11-3. Signal Processing Design Using Circuit Shown in [Figure 11-2](#)

To create signal processing designs for cosimulation:

1. To place the circuit subnetwork(s) you have already created in the signal processing schematic, choose Component > Component Library. Your opened projects are listed at the top of the list. Circuit projects have (A/RF) at the end.
2. Choose the circuit design you want and place it in your signal processing schematic.
3. Add the signal processing components.
4. Add the signal processing controller(s).
5. Connect the circuit design to the signal processing components.
6. For cosimulation with the Circuit Envelop simulator, see Circuit Simulation Controllers, later in this section.
7. If your circuit subnetworks have feedback loops between them, see Feedback Loops, later in this section.

8. If the input signal into the circuit subnetwork is not of type *Timed*, see *Numeric-to-Timed Converters*, below.
9. Start the simulation.

Circuit Simulation Controllers

As stated earlier, Agilent Ptolemy can cosimulate with only the Circuit Envelope or High-Frequency SPICE simulators. Any circuit simulation control components other than ENV or TRAN (such as for harmonic balance or S-parameter simulation) are ignored in the cosimulation from the signal processing schematic.

Numeric-to-Timed Converters

Both Circuit Envelope and Transient simulators deal with time-domain signals. Therefore, signal processing components connected to the circuit subnetwork need to be of type *Timed*. If the input component (connecting the signal processing components to the circuit) produces numeric data, place the appropriate numeric-to-timed converter (such as float-to-timed or complex-to-timed) in your schematic. These components are found in the Signal Converters library, and assure that the input into the circuit subnetwork is in the time domain. Refer to “[Time Converters](#)” on page 11-10 for more information on this topic.

Clustering of Circuit Subnetworks

Clustering is the process of defining the boundaries of the signal processing and analog/RF simulators. Initially, this boundary is defined by circuit schematics, where you define the circuit subnetworks and then make an instance of those on the Signal Processing schematic. However, there is a bit more to clustering than what is on the two schematics.

Circuit subnetworks directly connected in the Signal Processing schematic are automatically clustered by the program and treated as one circuit subnetwork, as

shown in [Figure 11-4](#). Therefore, use only one circuit simulation control component in either of the two (or more) directly connected subnetworks.

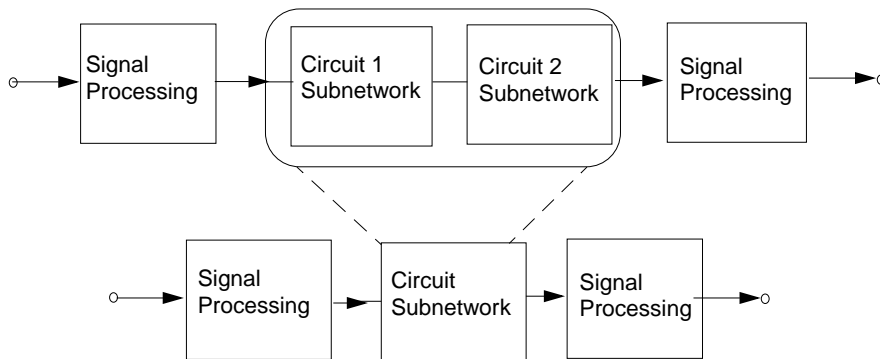


Figure 11-4. Connected Subnetworks Treated as One

Connected Circuit Subnetworks

For example, what happens when two circuit subnetworks defined on two different circuit schematics are connected on a Signal Processing schematic? The two circuit subnetworks are clustered into one by the program. This should not concern the user, since this would be done transparently. However, if each of these two circuit subnetworks use their own simulation controller, then the circuit engine would not know which one to choose for simulation. This would result in an error message.

Connected Resistors

Another aspect of clustering is when circuit components available on the Signal Processing schematic (resistors in the first release of Advanced Design System) are connected to a circuit subnetwork. In this case, such resistors will be absorbed into the circuit subnetwork during the clustering and will be simulated by the circuit engine as part of circuit subnetwork.

Feedback Loops

Circuit subnetworks that form a feedback loop via signal processing components require a delay component in the feedback loop to facilitate the signal processing simulation scheduling, as shown in [Figure 11-5](#) and [Figure 11-6](#). If such a delay is not

present, an error message will be issued. To have the program automatically insert the delay, you need to edit the parameters for the Data Flow controller. To do this, double-click the controller and choose the *Options* tab and then *Resolve deadlock by inserting tokens* from the Deadlock Management drop-down list. For more information about deadlocks, refer to “[Deadlocks](#)” on page 9-7 in [Chapter 9, Theory of Operation](#).

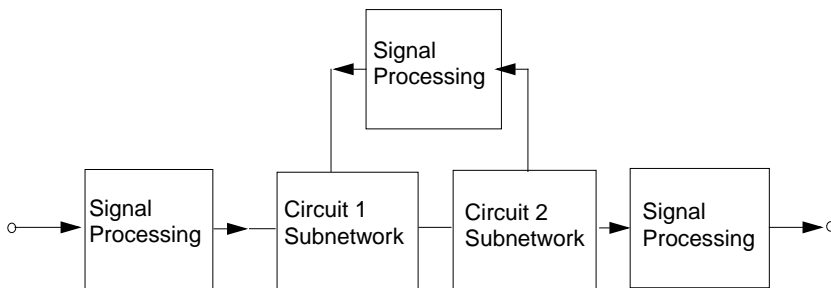


Figure 11-5. Feedback Loop Before Delay Added by Program

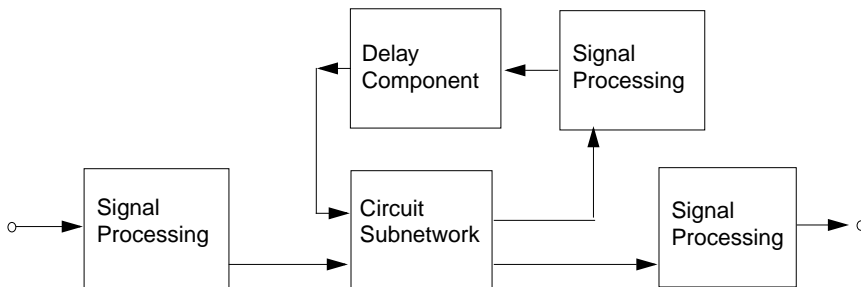


Figure 11-6. Feedback Loop After Delay Added by Program (Delay Not Shown on Schematic)

Named Connections and Measurements in Circuit Designs

Named connections and measurements included in the circuit schematic design (such as for a voltage) are ignored in cosimulation. The only results you get from a cosimulation are obtained from the Signal Processing schematic using Sink components or Interactive Components and Displays items.

Circuit Envelope Specific Rules

The output of the Circuit Envelope simulator is a collection of time waveforms, each at a different fundamental frequency. You need to select the waveform you want by specifying this fundamental frequency. You do this by choosing the *EnvOutSelector* or *EnvOutShort* component from the Circuit Cosimulation library. Refer to [“EnvOutSelector and EnvOutShort Components” on page 11-11](#) for more information. You should place this component at all circuit subnetwork output ports in the Signal Processing schematic.

Circuit Envelope simulation requires that many parameters be set up in the circuit schematic. For more information, refer to Advanced Design System circuit simulation documentation. For cosimulation, the key parameter is the *Step* parameter. This is the time step used by the simulator, and can be set equal to or less than the time step at the connecting port in the signal processing schematic design. Other important parameters for cosimulation (especially nonlinear designs) are *MaxOrder*, *Freq[]*, and *Order[]*. Make sure that the *OutFreq* parameter specified at the *EnvOutSelector* is among the fundamental frequency or harmonics specified by the Circuit Envelope controller.

Transient Simulation Specific Rules

When cosimulation with Agilent Ptolemy and the Transient simulator is required, the circuit schematic must have a Transient controller (also called a simulation component). No explicit user setting is required for the Transient controller. That is, the default parameters will work for cosimulation. However, the Transient controller's *Freq [x]* parameter is required whenever there are any frequency-dependent sources. The *Freq [x]* parameter specifies the fundamental frequency.

Nested Simulation Approach

Advanced Design System cosimulation is based on a nested simulation approach. In this use model, you first create your circuit designs on the circuit schematic. This circuit design can be tested using appropriate circuit sources and measurements with either the Circuit Envelope or High-Frequency SPICE simulators. Once the circuit design has been verified, ports to be interfaced with the signal processing design are identified and placed. Next, you place an instance of the design on the Signal

Processing schematic and connect it to the other blocks. The combined schematic design can now be simulated.

Signal Processing Model of the Circuit Network

Agilent Ptolemy uses a data flow simulation approach, and this simulation is controlled using the Data Flow Controller component. To understand this chapter, you need to be aware that this simulation is based on invoking a *schedule*. A schedule tells the simulator engine to “fire” components in a certain order and with a certain frequency. A simulation is typically a repetition of a schedule many times.

From the Data Flow engine perspective, a circuit subnetwork on the Signal Processing schematic is just a component with a certain number of input and output ports. This circuit subnetwork is part of the schedule determined by the Data Flow engine. It would be fired just like any other component according to the schedule, and as many times as required. Every time the circuit subnetwork is fired, the circuit simulator (designated by the simulation controller on the circuit schematic) continues to carry on the simulation based on the input it receives from the signal processing interface. Once the circuit simulator is finished with its analysis, it passes the simulation results back to the signal processing interface. This cycle repeats as many times as the scheduler requires. The duration of the circuit simulation each time it is invoked is determined by the time step provided by the connecting signal processing component at the input interface to the circuit subnetwork.

Circuit Model of the Signal Processing Network

From the circuit simulator engine point of view, the signal processing input interface is viewed as an ideal (0 ohm impedance) source. The more ports at the input interface to the circuit, the more ideal sources there will be feeding the circuit subnetwork. At the output interface of the circuit, there would be a node where the results are shared with the connecting signal processing component.

Interface Issues

At the interface boundary of the signal processing and analog/RF circuit simulators, there needs to be an exchange of information. The semantics and fundamentals of simulation in the two application areas are quite different and therefore, you need to understand these differences for proper use. The following sections outline the most important aspects of this interface.

Time Step

Time samples for signal processing are one fixed time step apart. However, both Envelope and Transient simulators define the time step in the simulation controller with various options.

The Transient Simulator controller component has several parameters, including Start time, Stop time, Min time step, and Max time step (see the Time Setup tab). In addition, the Integration tab contains a time step control method parameter with Fixed, Iteration Count, and Truncation Error options. For more information, refer to the RF Transient/Convolution Simulation chapter in the *Circuit Simulation* manual.

For cosimulation with the Transient simulator, keep in mind one key issue: The Transient simulator may need time steps smaller than Agilent Ptolemy's Time Step to satisfy its own setup requirements. In addition, the Transient simulator, when needed, will take additional time steps to match the time points in the signal processing simulation. Only time steps that match the signal processing time points will be passed on to Agilent Ptolemy.

Note For all practical purposes, the only parameter that may concern the cosimulation user is the Max time step. The rest of the parameters in the Transient Simulation control component can be left at default value.

For the Circuit Envelope simulator, the time step parameter in the ENV Simulation controller component should be set equal to or less than the Time Step at the signal-processing-to-circuit interface.

Setting the Analog/RF subnetwork time step for Circuit Envelope (Envelope) or Transient (Tran) cosimulation: The Envelope controller parameter *Time step* or the Tran controller parameter *Max time step* should be set equal to a submultiple of the DSP Schematic DataFlow controller (DF) simulation time step. Call this *tstep*. You can set this value and pass it into the Analog/RF circuit with *Time step* or *Max time step* set equal to $tstep/N$, where N is a user-defined integer typically greater than 1. Set $N > 1$ when necessary, such as when the Envelope or Tran simulation signal spectrum is important, or when accuracy issues are important.

Note The Stop time for the simulation is determined by the Signal Processing's Data Flow controller and/or Sinks. The Circuit Envelope or Transient component's

Stop time is not relevant.

Delays in Feedback Loops

As stated earlier, Data Flow simulation requires that a Delay component exist in the feedback loops for proper activation of the schedule. Circuit subnetworks that form a feedback loop, for this same reason require a delay component in their path.

Typically, a Delay_Rf component in such feedback loops will suffice. If such a delay does not exist, Agilent Ptolemy will report a deadlock by default.

Time Converters

The common signal being exchanged between signal processing Data Flow components and the circuit simulators (Circuit Envelope and Transient) is a time-domain signal. All three engines, hence, deal with the notion of time step.

The signal entering the circuit subnetwork should be Timed. The Transient simulator deals only with real-baseband time-domain signals while Circuit Envelope can handle both baseband and complex envelope timed signals.

If the signal entering into the circuit subnetwork is not Timed (that is, the signal is Numeric), you should place a FloatToTimed, FixedToTimed, IntToTimed, or CxToTimed converter to accommodate the conversion. Although Agilent Ptolemy will place appropriate converters when they do not exist, it is *always* a good practice to explicitly place and connect these converters in your design. This will ensure that the input parameters into the circuit subnetwork are correct, as well as helping to debug possible errors that might occur.

Carrier Frequency

In the case of cosimulation with the Circuit Envelope simulator, the timed signal entering the circuit subnetwork is typically a carrier-modulated timed signal. This means that timed data has an F_c field that is passed to the Circuit Envelope simulator, which is needed by the simulator. The Circuit Envelope simulator, depending on a particular design, will generate a number of time-domain waveforms, each associated with a carrier (harmonic) frequency. Since Agilent Ptolemy supports only *one* carrier frequency at each node, you need to select which one of the waveforms you desire in the signal processing portion of the design. This is done by placing a Circuit-Envelope specific component described next.

EnvOutSelector and EnvOutShort Components

When cosimulating with the Circuit Envelope simulator, additional information is needed for proper cosimulation. This is done by connecting an EnvOutSelector or EnvOutShort component (from the Circuit Cosimulation library in the Signal Processing schematic) to each output port of the subnetwork design.

The EnvOutSelector component acts as an open, blocking everything connected to its output from loading the circuit. If such loading is desired, use the EnvOutShort component. The EnvOutShort component acts as a short and therefore loads the circuit with the connecting Signal Processing components.

The EnvOutSelector and EnvOutShort components have a parameter called *OutFreq*. OutFreq specifies which waveform is selected from the time-domain waveforms at the output of the Circuit Envelope simulator. OutFreq has the following options:

- Lowpass—selects the time-varying DC component.
- Bandpass—(default) lets you specify any frequency.
- Allpass—forms the composite (baseband) signal.

One or more EnvOutSelector components can be connected to each output port of a circuit subnetwork, refer to [Figure 11-7](#), below. This means that all waveforms generated by the Circuit Envelope simulator can be accessed in the Signal Processing schematic.

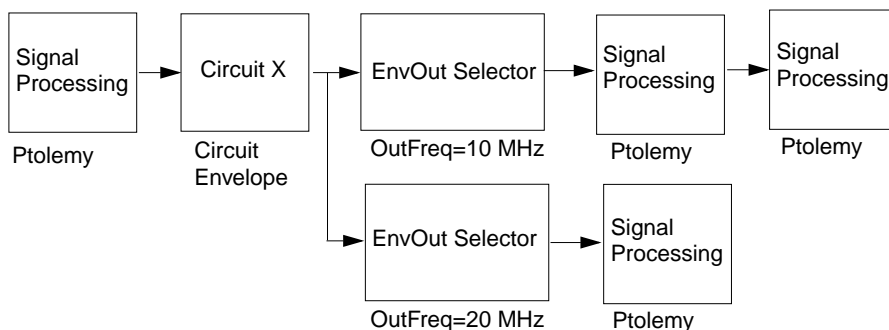


Figure 11-7. *EnvOutSelector* Components at each Circuit Subnetwork Output Port

If either the EnvOutSelector or EnvOutShort component is used with a design simulated by the Transient simulator, their effect is to be either an open or a short, respectively. Other than this, they do not affect transient cosimulation designs, so

this component can be left in place without any impact on the cosimulation with the Transient simulator.

Snapping Rule

In the Bandpass option of the OutFreq parameter, you can type in the desired fundamental whose time waveform you are interested in. If the frequency you specify does not exist in the list of fundamentals, the interface program will search and snap to the nearest fundamental. Presently, anything within 0.01% of a fundamental will be snapped to that fundamental frequency. If the frequency specified in the Bandpass option of OutFreq is not within 0.01% of the fundamental, a default value of 100 MHz will be used and a warning message issued.

Troubleshooting Common Problems

Although the cosimulation use model is intuitive, there are a few pitfalls you should know about to avoid errors:

1. At present in Advanced Design System, only the Transient and Circuit Envelope circuit simulators can cosimulate with Agilent Ptolemy. Other circuit simulation controllers on the analog/RF schematic (such as S-parameter or AC) will be ignored in cosimulation.
2. Directly connected circuit subnetworks placed as instances on Signal Processing schematics are clustered together and hence should be considered as one circuit subnetwork. This means that if each of these subnetworks has their own circuit simulation controller, an error message will be issued. To avoid such problems you can either:
 - Deactivate all controllers but one on the circuit schematics.
 - Connect a signal processing component between the two circuit subnetworks, thereby preventing the two subnetworks from being clustered into one.
3. Resistor components that are part of hierarchical designs of timed components in the Signal Processing schematic will be absorbed into connecting circuit subnetworks by the program. If the EnvOutSelector component is used, the absorbed resistors will *not* load the circuit, since the circuit model is an open. If the EnvOutShort component is used, the absorbed resistors will load the circuit, and the results will be different by a scale factor.
4. Since resistors that are part of timed subnetworks are absorbed into connecting circuit subnetworks, you should avoid placing a sink or any other signal

processing component directly at this port, when cosimulating with Circuit Envelope. If placed, an error message is issued, requiring an EnvOutSelector component to be placed. The reason for this condition is the fact that the sink now constitutes an output port from the perspective of the circuit subnetwork.

Cosimulation Example

To illustrate cosimulation, we will use an example called RectifierCosim_prj.

Copying and Opening the Project

1. From the Main window, choose **File > Copy Project**. A dialog box appears.

Note On UNIX platforms, you will not be able to work in the Advanced Design System Examples directories. You must copy the example project to a directory for which you have write permission. On PC platforms, while you can work in the Examples directories if you want, it's still a good idea to copy the examples to another directory.

2. In the From Project field, click the **Examples Directory** button, and then the **Browse** button. The File Browse dialog box appears with the Example directories listed.
3. Select the **/Com_Sys** directory.
4. Select **RectifierCosim_prj** from the list of files in the Files field.
5. In the To Project field, click the **Startup Directory** or **Working Directory** button (depending on where you want to copy the project to) or choose the **Browse** button if you want to select another directory.
6. Choose **Copy Project Hierarchy**. This ensures that all the appropriate directories and files will be copied.
7. Click **OK** to copy the project and close the dialog box.
8. From the Main window, choose **File > Open Project**. When the Open Project dialog box appears, select **<the directory you copied the example to>** in the Directories field, then double-click **RectifierCosim_prj** in the Files field.

Rectifier Schematic

The top level design RectifierCx_Tutorial, as shown in [Figure 11-8](#), generates a complex modulated signal and sends that signal into a simple rectifier circuit.

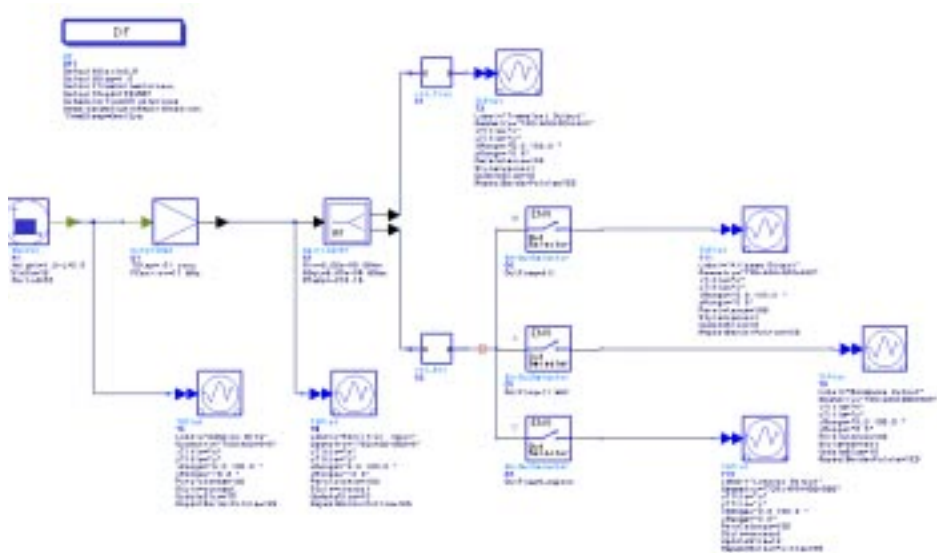


Figure 11-8. RectifierCosim Project Top-Level Schematic

Two identical instances of this rectifier circuit are created on a circuit schematic, one with a TRAN controller (rct_Tran), [Figure 11-9](#), and the other with an ENV controller (rct_Env), [Figure 11-10](#).

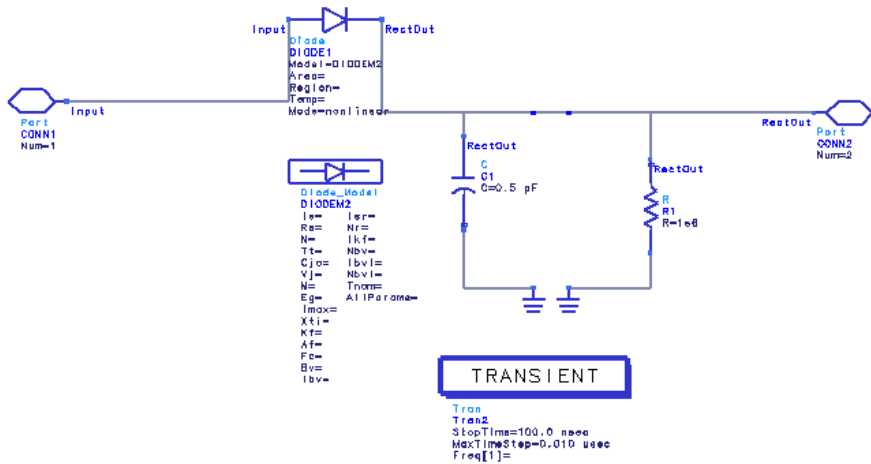


Figure 11-9. Circuit Subnetwork with Transient Controller

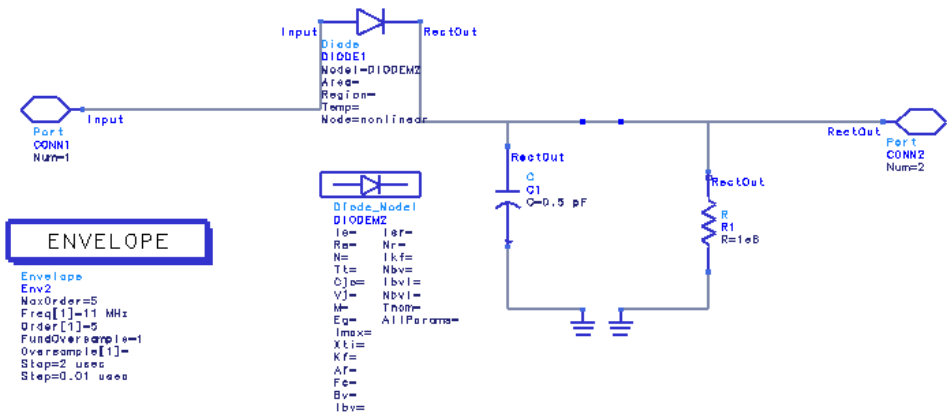


Figure 11-10. Circuit Subnetwork with Circuit Envelope Controller

To view the circuit subnetworks from the top-level design, choose *View > Push Into Hierarchy* or click the *Push Into Hierarchy* button (down arrow) from the toolbar.

The circuit design in both *rct_Tran* and *rct_Env* is a simple diode with a shunt parallel RC attached to its output. Note the placement of ports and that *rct_Env* has an Circuit Envelope controller, while *rct_Tran* a Transient controller. The two circuit designs are then placed on the Signal Processing schematic.

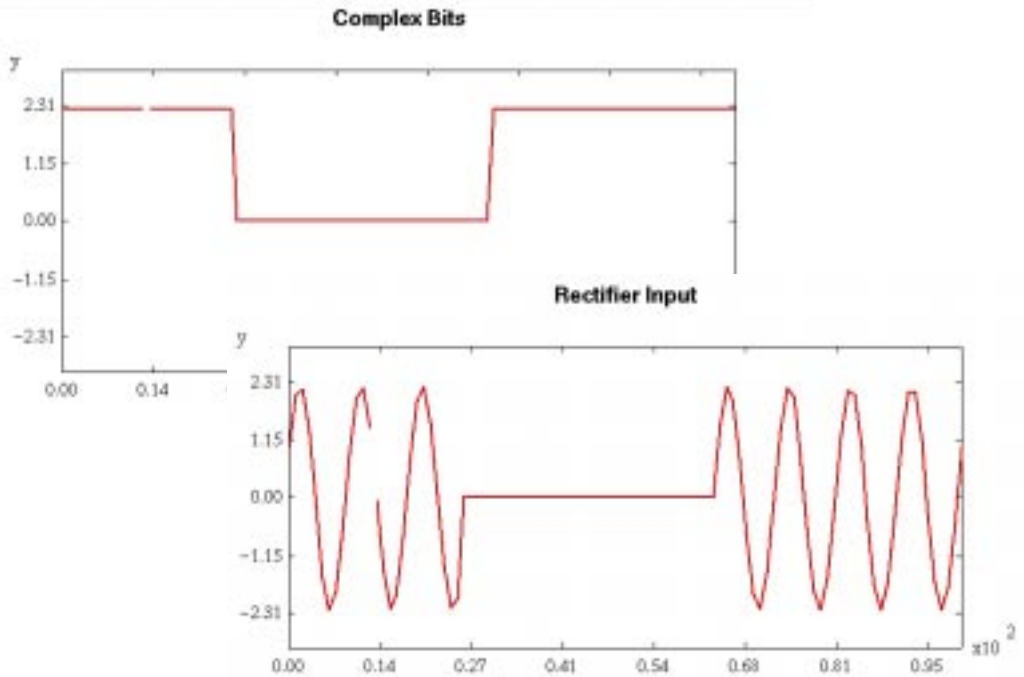
On the Signal Processing schematic, the generation of complex modulated signals is accomplished via a RectCx component that generates a periodic pulse with a complex amplitude. This complex pulse is then fed into a CxToTimed component that effectively upconverts the signal. The TStep is set to 0.01 μ sec and the carrier is at 11 MHz. This modulated RF signal is then split into two branches by a SplitterRF component and fed into the rct_Tran and rct_Env circuit subnetworks. In addition, there are TkPlot components (which display the simulation results) attached to the output of RectCx and CxToTimed to monitor the signal before simulation.

Since the TStep of the signal entering circuit design is at 0.01 μ sec, the MaxTimeStep on the Transient controller is set to the same value. This value should always be smaller than or equal to the Signal Processing TStep. The other Time setup parameters, such as Start time and Stop time, are ignored in the Transient cosimulation. Note that the output of rct_Tran is directly fed into a TkPlot without an interface component.

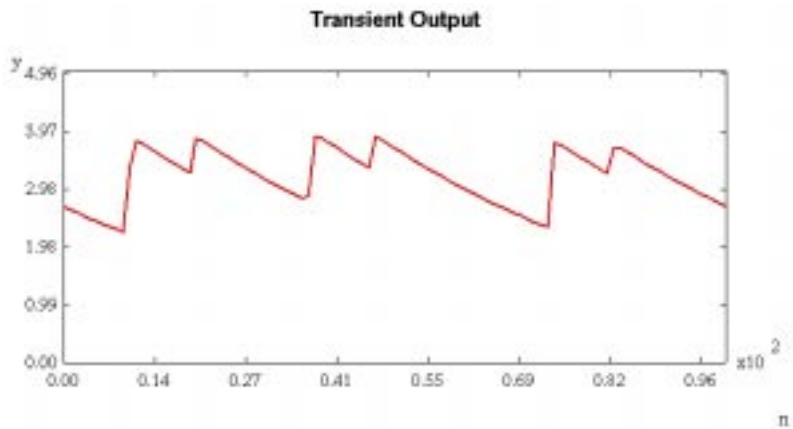
Similarly for the Circuit Envelope simulator, the Step parameter of the simulation controller should be set less than or equal to the Signal Processing TStep. Other parameters of interest are Freq[], Order[], and MaxOrder, which specify the fundamentals and related harmonics to be analyzed. In this example, the fundamental of interest is Freq[1] = 11MHz and the MaxOrder and Order[1] are set to 5. Note also, that Freq[0] is the dc term that is always available.

Typically, there is only one EnvOutSelector component attached to the Circuit Envelope subnetwork output, but in this example we have used three to show the different signals that can be selected from the Circuit Envelope output. Specifically, the OutFreq parameter is set to the Bandpass, Allpass and Lowpass options in the three instances. When the Bandpass option is selected, the dialog box changes so you can enter the desired fundamental frequency. In this case, OutFreq is set to 11 MHz. The output of EnvOutSelectors are then fed into three interactive TkPlot display components.

When we simulate this design, a total of six TkPlot windows pop up, as shown on the following three pages.



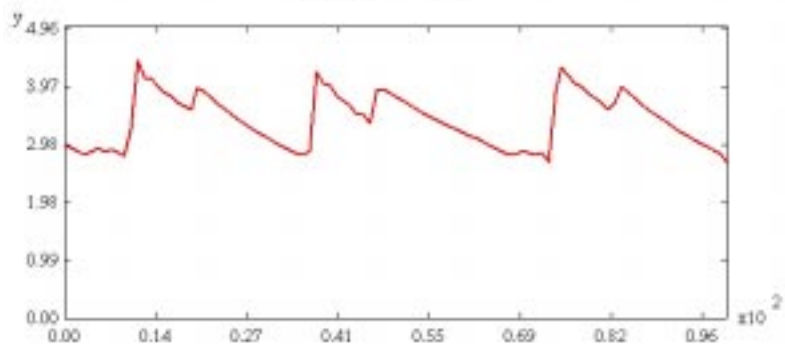
The plot *Complex Bits* displays the magnitude of the complex periodic pulse and the plot *Rectifier Input* depicts the pulse-modulated signal at 11 MHz.



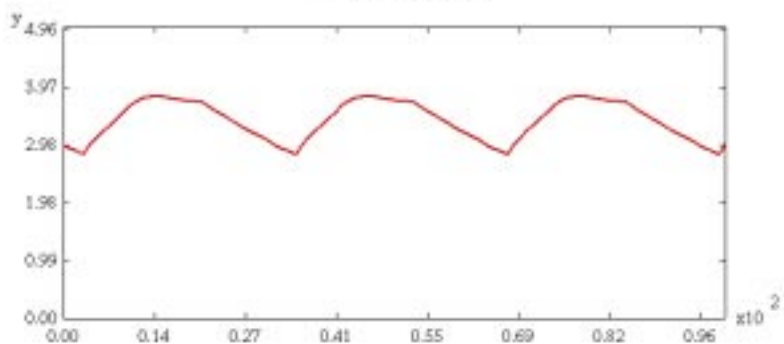
The plot *Transient Output* is the rectified version of the modulated signal (note that there are no negative components in the signal), where the value of the time constant (RC) determines the degree of pulse fall-off. Note also that this output is a real-baseband signal and includes all the harmonics.

On the next page, the Allpass, Bandpass, and Lowpass output plots are shown.

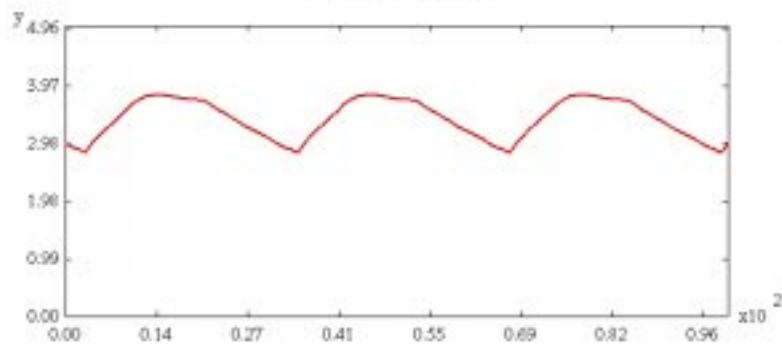
Allpass Output



Lowpass Output



Bandpass Output



Chapter 12: Using Interactive Controls and Displays

Agilent Ptolemy contains a library of 19 Interactive Control and Display components. These components provide real-time simulation input control and animated plots of your simulation results. Interactive Displays represent one of two general use models for viewing simulation results.

In contrast, the main Advanced Design System method to display simulation results is to save your simulation data and open a Data Display window to view the results. You can set up your schematic for both methods, by placing both an Interactive Control and Display component and a Sink. The following table describes the two basic use models for displaying results:

Table 12-1. Comparison of Simulation Results Methods

Method	Description
Interactive Controls and Displays	Animated displays of simulation results and interactive control items. Data is not saved. No post processing. Display starts during simulation.
Data Display Window	Data is saved after simulation is complete. You open a Data Display window, choose a plot type, parameters to plot, and have many data manipulation options.

The Interactive Control and Display components are a quick and easy way to display your simulation results. They also give you a way to interactively change parameters while your simulation is running and display animated plots.

The Interactive Controls and Displays were derived from a scripting language called Tool Command Language (Tcl) and the Tool Kit attached to Tcl (Tk). These were originally developed at the University of California at Berkeley. Therefore, the abbreviations Tcl or Tk appear in the component names.

Note To use the Interactive Controls and Displays library components (such as TkPlot) with Advanced Design System's tune mode, you must dismiss the Interactive Controls and Displays component between each tune with its pop-up dialog box.

List of Interactive Control and Display Components

Table 12-2 shows the Interactive Controls and Displays library.

Table 12-2. Interactive Controls and Displays Library

Descriptive Name	Component Name
Interactive Slider	TkSlider
Plot Inputs versus Time	TkPlot
Display History of Input Values	TkText
Input Values Display	TkShowValues
Plot Y versus X Inputs	TkXYPlot
Bar Chart Display	TkBarGraph
Interactive Complex LMS Adaptive Filter	LMS_CxTkPlot
Interactive LMS Adaptive Filter	LMS_TkPlot
Interactive Buttons	TkButtons
Conditional Breakpoint	TkBreakPt
Bar Meters Display	TkMeter
Booleans Display	TkShowBooleans
Baseband Equivalent Channel	TkBasebandEquivChannel
Invoke Tcl Script	TclScript
Eye Diagram	TkEye
IQ Constellation Diagram	TkConstellation
Histogram Diagram	TkHistogram
Display rms value of input IQ signal	TkIQrms
Signal Power Display in dBm	TkPower

In this chapter, you will find:

- An introduction to the Interactive Control and Display components.
- Cross-references to the *Signal Processing Components* manual, which contains a description of the parameters associated with each component.
- References to application examples shipped with Advanced Design System that contain or illustrate various Interactive Control and Display components.

TkSlider and TkPlot Components Example

The following figure is taken from the `./examples/DSP/dsp_demos_prj` project. It is called EYE. If you want to run this example, copy the project from the examples directory to a directory for which you have write permission, using the *File > Copy Project* command.

The EYE schematic uses an Interactive Control component called TkSlider to adjust the amount of noise added to the pulse stream. You then can instantly see the simulation results change in an animated display provided by the TkPlot component. TkPlot simply plots one input on the Y-axis versus time, or sample number, on the X-axis.

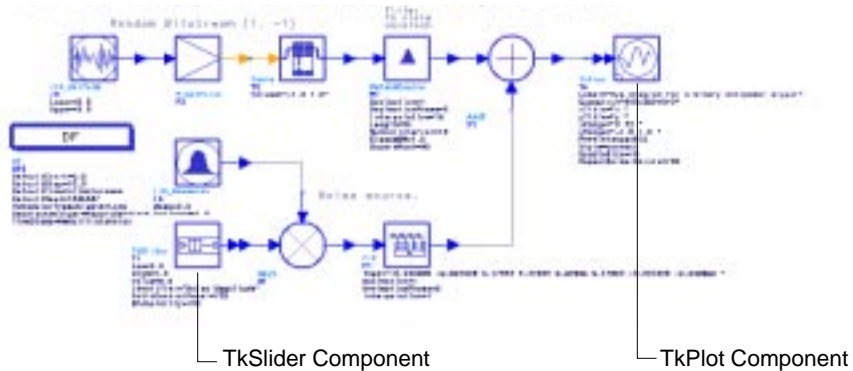


Figure 12-1. Schematic Using TkSlider and TkPlot Components

When you begin simulation of a design using TkSlider, the dialog box shown in [Figure 12-2](#) is displayed.

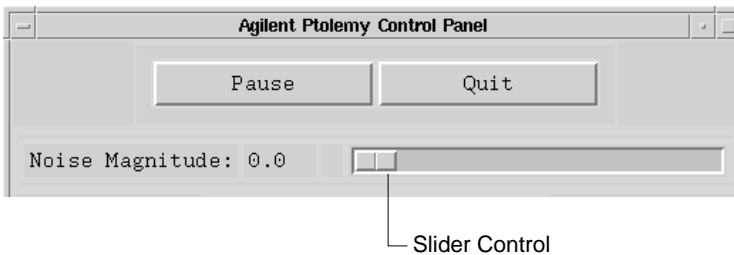


Figure 12-2. TkSlider is Interactively Controlled by This Dialog Box

This dialog box is called the Agilent Ptolemy Control Panel. At a minimum, the Pause and Quit buttons are shown. One of the parameters for TKSlider is called PutInControlPanel. If the default of Yes is accepted, the slider is put in the Control Panel. (If you choose No for this parameter, a separate box will be displayed for each slider.)

With the slider control all the way to the left, 0.0 noise is added and the resulting clean eye-diagram plot is shown below.

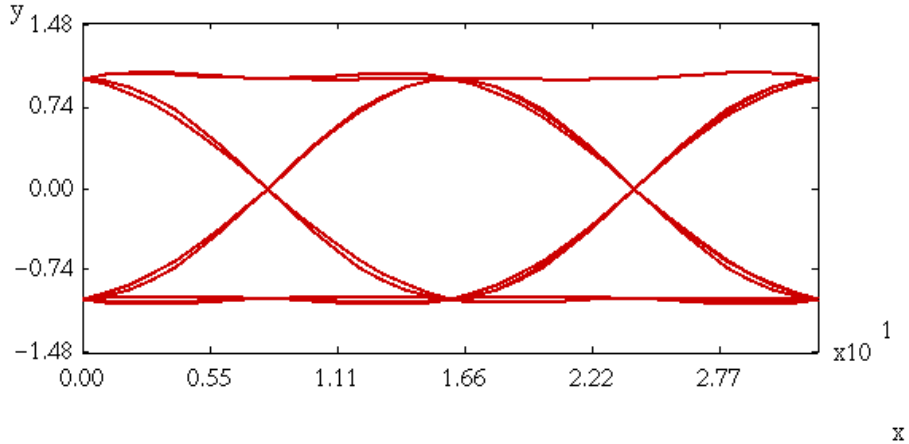


Figure 12-3. Eye Diagram Plot with Noise at 0.0

The slider control can be adjusted, as shown in the next figure.

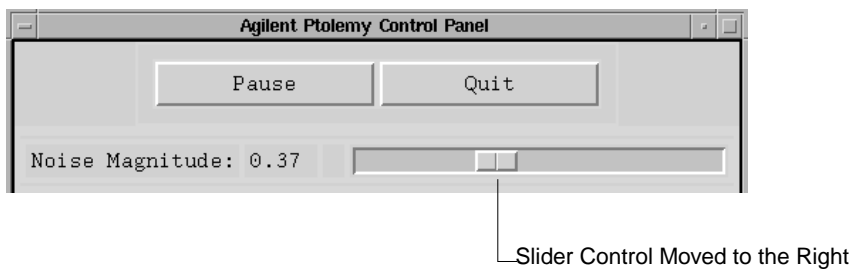


Figure 12-4. Slider Control Moved to Add Noise

If you move the slider to the right so that the Noise Magnitude is 0.37, the animated plot changes to display the poor eye diagram shown in [Figure 12-5](#).

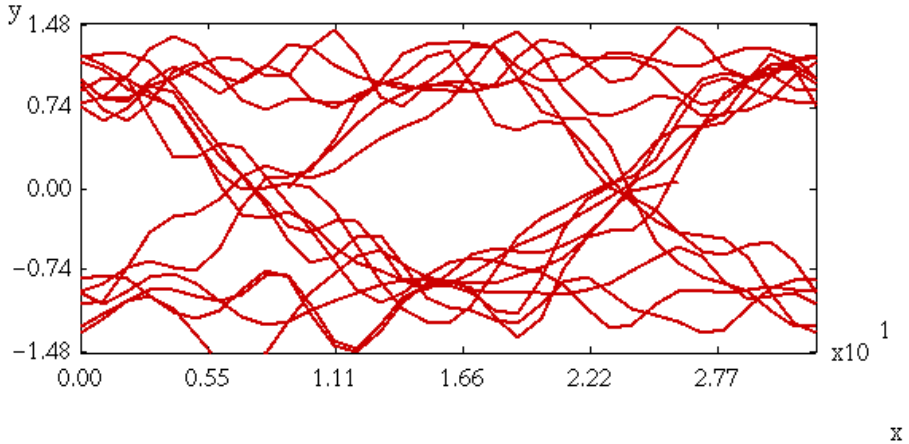


Figure 12-5. Eye Diagram Plot with Noise at 0.37

More or less noise can be added and the results will be shown immediately.

For complete parameter description, see *TkSlider (Interactive Slider)* or *TkPlot (Plot Inputs versus Time)* in the *Signal Processing Components* manual.

Placing Multiple TkSlider Components

Placing multiple TKSlider components in a schematic will result in multiple sliders in the Control Panel, as shown below. You can type a label for each by editing the

parameters on-screen or double-clicking the component to bring up the Component Parameters dialog box.

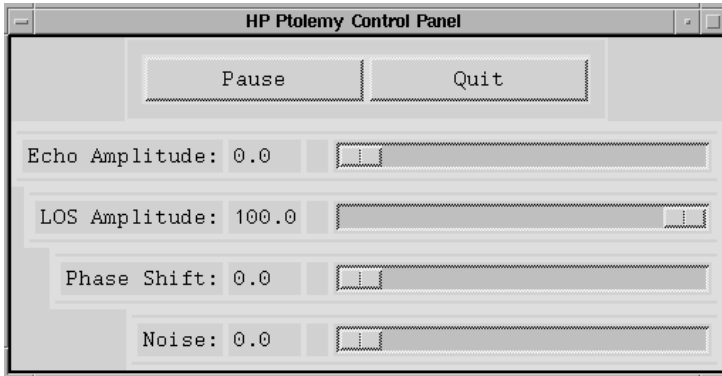
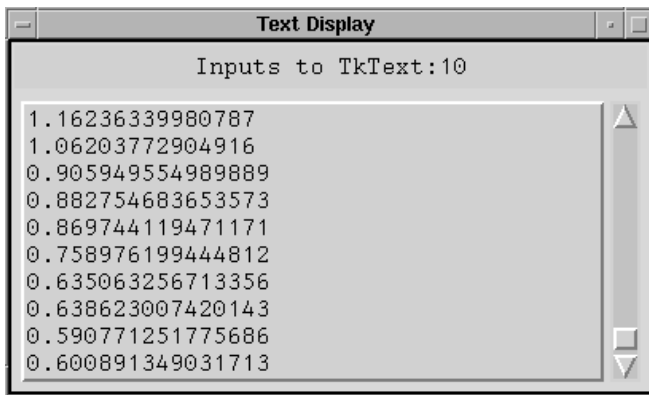


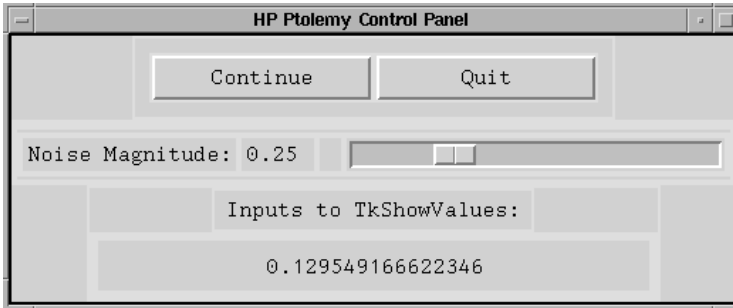
Figure 12-6. Control Panel with Multiple TkSliders

TkText and TkShowValues

The TkText component simply displays the input values in text form, as shown below. This is an input history.



The TkShowValues component works in a similar manner, except only one value is shown at a time (rather than a history). This value is displayed in the lower part of the Agilent Ptolemy Control Panel, as shown below.



For complete parameter description, see *TkText (Display History of Input Values)* or *TkShowValues (Input Values Display)* in the *Signal Processing Components* manual.

TkXYPlot Component

The following figure is taken from the `./examples/DSP/dsp_demos_prj` project. It is called EQ_16QAM. If you want to run this example, copy the project from the examples directory to a directory for which you have write permission, using the *File > Copy Project* command.

Several TkXYPlot components are placed in this design to show animated results where two inputs, X versus Y, are to be plotted. In this example the TkXYPlot

components are used to display constellation diagrams. The equalized constellation diagram is shown in [Figure 12-8](#).

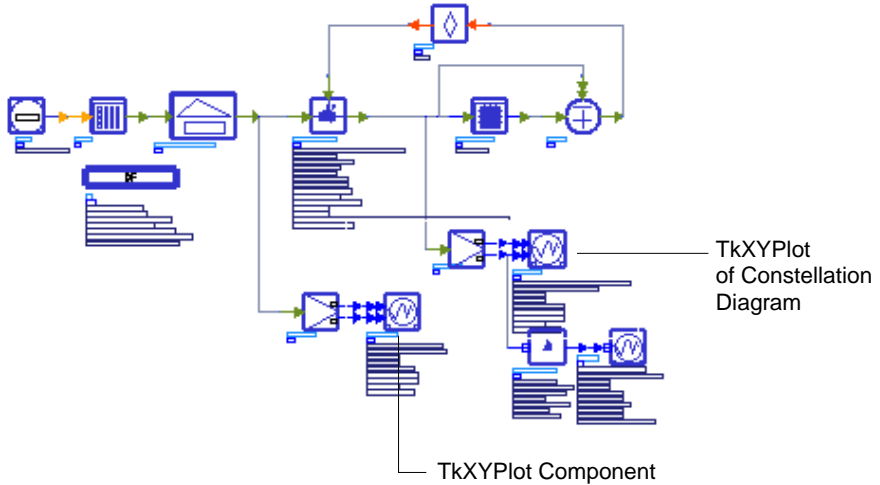


Figure 12-7. Equalized 16 QAM System with Multipath and Phase Noise.

For complete parameter description, see *TkXYPlot (Plot Y versus X Inputs)* in the *Signal Processing Components* manual.

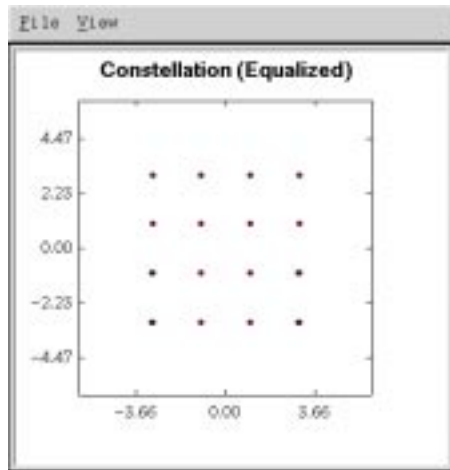


Figure 12-8. TkXYPlot for Equalized Constellation Diagram

TkBarGraph

The TkBarGraph component displays input (multiple anytype) data in a bar graph format. Different inputs are assigned different colors, up to 12, then the colors are repeated. A TkBarGraph example is shown in [Figure 12-9](#).

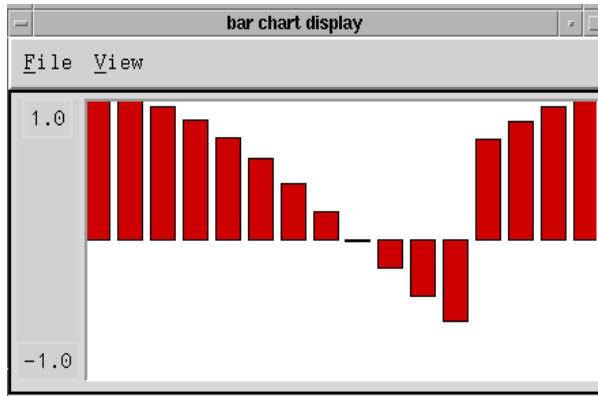


Figure 12-9. TkBarGraph Display

For complete parameter description, see *TkBarGraph (Bar Chart Display)* in the *Signal Processing Components* manual.

LMS Adaptive Filter Components

There are two LMS adaptive filter components in the Interactive Controls and Displays library: *LMS_CxTkPlot* and *LMS_TkPlot*. They are the same except that the former expects complex data and the later expects real data.

An example of this component can be found in the same project we have just used: `../examples/DSP/dsp_demos_prj` project. It is called *EQ_16QAM*, and is shown in [Figure 12-10](#).

Both LMS components implement an adaptive filter using the least-mean square algorithm, also known as the stochastic-gradient algorithm.

The size of the LMS filter is determined by the number of coefficients in the *Taps* parameter; the default gives an 8th-order, linear-phase lowpass filter. LMS supports decimation, but not interpolation.

The filter coefficients can be specified directly or read from a file. To load filter coefficients from a file, replace the default coefficients with the string *<filename>* (use an absolute path name for the filename to allow the filter to work as expected regardless of the directory where the simulation process actually runs).

When used correctly, this LMS adaptive filter will adapt to try to minimize the mean-squared error of the signal at its error input. The output of the filter should be compared to (subtracted from) some reference signal to produce an error signal. That error signal should be fed back to the error input. The ErrorDelay parameter must equal the total number of delays in the path from the output of the filter back to the error input. This ensures correct alignment of the adaptation algorithm. The number of delays must be greater than 0 or the simulation will deadlock.

If the SaveTapsFile string is non-null, a file will be created with the name given by that string, and the final tap values will be stored there after the run has completed.

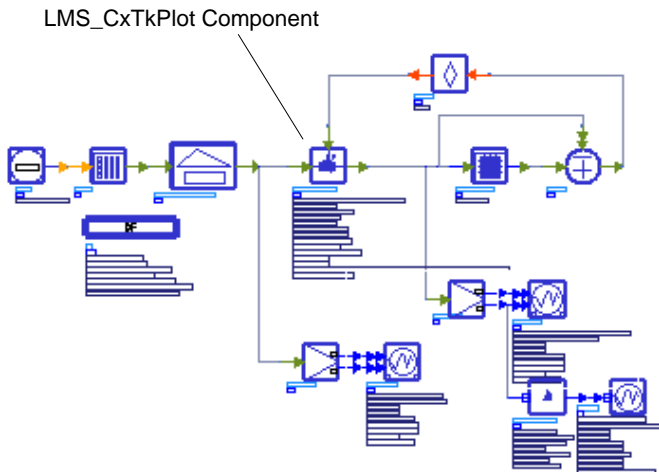


Figure 12-10. LMS Adaptive Filter (Complex) Component in the 16 QAM System

The plot generated from this design upon simulation is shown in [Figure 12-11](#).

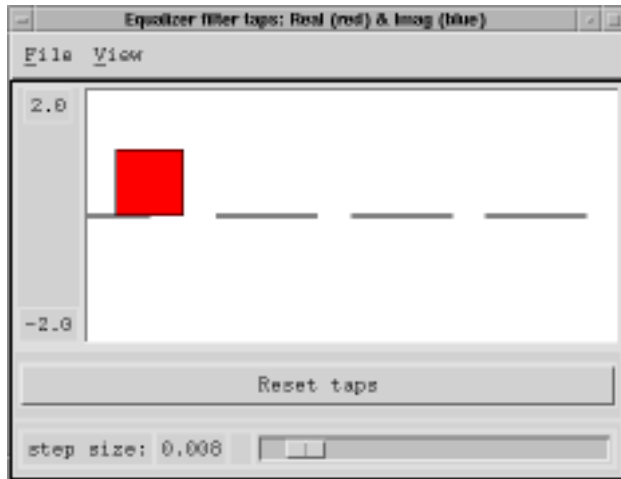
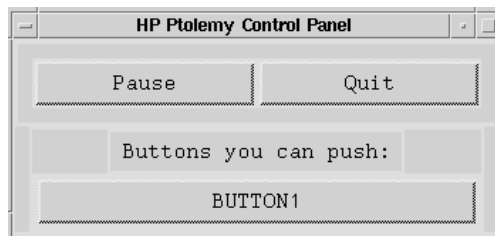


Figure 12-11. Plot Resulting From LMS_CxTkPlot FilterTaps Parameter

For complete parameter description, see *LMS_CxTkPlot (Interactive Complex LMS Adaptive Filter)* or *LMS_TkPlot (Interactive LMS Adaptive Filter)* in the *Signal Processing Components* manual.

TkButtons

Like TkSlider, TkButtons produces an output. The data type of the output is multiple anytype. TkButtons outputs 0.0 unless the corresponding button is pushed, when the output becomes the value assigned in the parameter *Value*. You can assign your own identifiers using strings for the corresponding parameter.



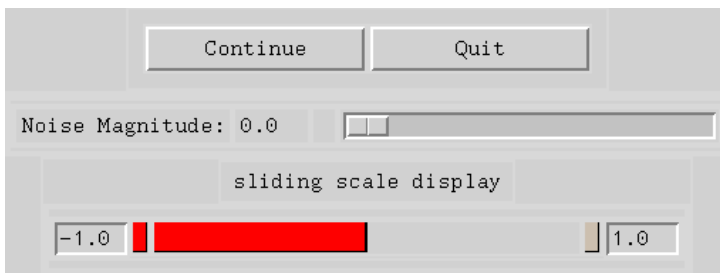
For complete parameter description, see *TkButtons (Interactive Buttons)* in the *Signal Processing Components* manual.

TkBreakPt

TkBreakPt allows you to pause or stop a simulation. Its principal use is to help debug simulations. You can stop the simulation based on a condition of the model's inputs. The parameter syntax is described in *TkBreakPt (Conditional Breakpoint)* in the *Signal Processing Components* manual.

TkMeter

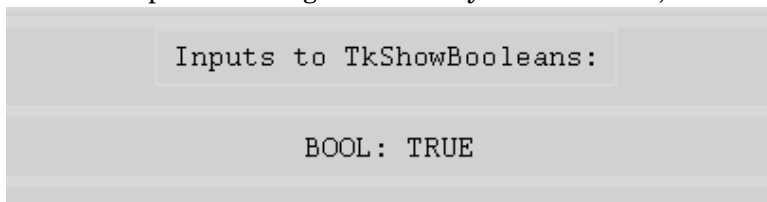
TkMeter dynamically displays the value of any number of input signals on a set of bar meters. The input type is anytype. These values are displayed in the lower part of the Agilent Ptolemy Control Panel, as shown below.



For complete parameter description, see *TkMeter (Bar Meters Display)* in the *Signal Processing Components* manual.

TkShowBooleans

The TkShowBooleans component works in a similar manner to TkShowValues, except that a Boolean value of zero (false) or non-zero (true) is shown. This value is displayed in the lower part of the Agilent Ptolemy Control Panel, as shown below.



For complete parameter description, see *TkShowBooleans (Booleans Display)* in the *Signal Processing Components* manual.

Baseband Equivalent Channel

The `TkBasebandEquivChannel` component is a subnetwork as shown in [Figure 12-12](#). It models a baseband equivalent channel with linear distortion, frequency offset, phase jitter, and additive white Gaussian noise. You can dynamically set many of the channel model parameters. `TkBasebandEquivChannel` accepts complex data and outputs the input signal plus distortions.

To model linear distortion, such as intersymbol interference, the input signal is passed through a complex FIR filter with the taps set by `LinearDistortionTaps`. The frequency offset distortion is set by the `Freq. Offset` slider control. Similarly, the phase jitter amplitude (peak-to-peak, in degrees) is set by the `Phase Jitter` slider control while the phase jitter frequency is set by the `PhaseJitterFrequencyHz` parameter. The phase of both the frequency offset and the phase jitter can be reset with the `Reset Phase` control button. The amplitude of the added complex white Gaussian noise is set by the `Noise Power` slider control.

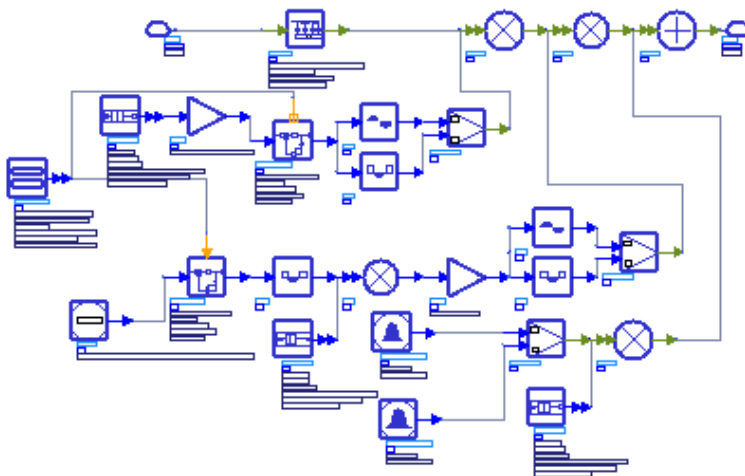


Figure 12-12. `TkBasebandEquivChannel` Component Subnetwork

For complete parameter description, see *TkBasebandEquivChannel (Baseband Equivalent Channel)* in the *Signal Processing Components* manual.

Invoke Tcl Script

TclScript reads a file containing Tcl commands. It can be used in a variety of ways, including using Tk to animate or control a simulation. A number of procedures and global variables will have been defined for use by the Tcl script by the time it is sourced. These enable the script to read the inputs to the component or set output values. The Tcl script can optionally define a procedure to be called by Agilent Ptolemy for every simulation of the component.

The parameter syntax is described in *TclScript (Invoke Tcl Script)* in the *Signal Processing Components* manual.

TkEye, TkConstellation, TkHistogram, TkIQRms, and TkPower

These TclTk components generate an eye diagram, an IQ constellation diagram, a histogram diagram, a display of the rms value of the IQ input signal, and a signal power display in dBm, respectively. For more information on these components, refer to *Interactive Controls and Displays* in the *Signal Processing Components* manual.

Additional Resources on Tcl/Tk

As stated earlier, the Interactive Controls and Displays Library was created using Tcl/Tk. Some users may want to explore this language further to write their own applications.

Tcl is a language that was created to be easily embedded into applications. Tk is a graphical toolkit that makes creating user interfaces easier. Both were created by John Ousterhout while he was a professor at U.C. Berkeley.

Books

- [1] John K. Ousterhout, *Tcl and the Tk Toolkit (Addison-Wesley Professional Computing)*, Addison-Wesley Publishing Company. May 1994.
- [2] Brent B. Welch, *Practical Programming in Tcl & Tk*, Prentice Hall: Englewood Cliffs, NJ. July 1997. 2nd Bk and cdr Edition.

[3] Mark Harrison and Michael J. McLennan, *Effective Tcl/Tk Programming: Writing Better Programs in Tcl and Tk*, Addison-Wesley Publishing Company. November 25, 1997.

World Wide Web

The following Web URL is a good place to start looking for information on Tcl/Tk:

http://www.yahoo.com/Computers_and_Internet/Programming_Languages/Tcl_Tk/

Chapter 13: 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 14, 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 13-1](#) to see some of 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 Refer to the Glossary for unique UC Berkeley Ptolemy terms, such as star and particle.

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, as follows:

Table 13-1. Compiler Requirements for Model Development

Operating System	Compiler
HP-UX 10.2x, 11	HP aC++ Version A.01.23 or higher and HP C/ANSI C Compiler A.10.32 or higher ¹
SunOS 5.7, or 5.8 Solaris 7.0, or 8.0	SPARC compiler Forte Workshop 6 or C++ Version 5.x or higher ²
Windows 98, NT 4.0, or 2000	Microsoft Visual C++, Professional Edition, Version 6 ²
IBM AIX 4.2 or higher	VisualAge C++ for AIX Version 5.0 ²

¹ The C compiler is needed in cases where external C files or routines will be compiled and linked in to the C++ Agilent Ptolemy model.

² An ANSI C Compiler is included with this compiler and will work for both the Agilent Ptolemy and the ADS Analog Model Development Kit.

Note The default environment size under Windows 98 *only* (not NT/2000) is insufficient for building models. There are two ways you can fix this problem:

Method 1: Increase memory by adding the following line to your C:\CONFIG.SYS file and rebooting:

```
SHELL=C:\COMMAND.COM /E:4096 /P
```

Method 2: First right-click on the MS-DOS icon and then choose Properties. Next, choose the Memory tab and type or click 4069 in the Initial Environment box. Finally, click OK.

Note that although the 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

You should first set the HPEESOF_DIR environment variable to wherever you've installed ADS.

Second, add the \$HPEESOF_DIR/bin, or %HPEESOF_DIR%\bin under Windows, directory to your PATH.

Under Windows, you can simply set these variables from an MS-DOS window with the Set command.

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.

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:

```
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:

```
cd hptolemy/src
cp $HPEESOF_DIR/doc/sp_items/SDFSin.pl SDFMySin.pl
vi SDFMySin.pl
```

and in Windows:

```
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:

```
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

```
PL_SRCS = SDFMySin.pl
```

If you are using the SPARC Compiler version 5 or higher, 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 -v`.

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:

```
cd ~/hptolemy
hpeesofmake "debug=1"
```

and in Windows:

```
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.

Note 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 13-3](#), 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 13-12](#) 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:

```
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,

```
INCLUDEPATH += /libtree/headers
SRCS += myutilities.c moreutilities.cxx
OBJS += /libtree/objs/tree$(OBJSUFFIX)
# OBJSUFFIX 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:

Table 13-2. Dependencies

Directory	make-defs Variable
<code>hptolemy-kernel/compat</code>	(always included)
<code>hptolemy-kernel/kernel</code>	<code>KERNEL</code>
<code>numeric/kernel</code>	<code>SDFKERNEL</code>

Table 13-2. Dependencies

Directory	make-defs Variable
numeric/base/stars	SDFSTARS
numeric/dsp/stars	SDFDSP
numeric/libptdsp	PTDSP
timed/kernel	TSDFKERNEL
timed/base/stars	TSDFSTARS
matrix/base/stars	SDFMATRIX
fixpt-analysis/base/stars	SDFFIX
controls-displays/tcltk/ptklib	PTK
controls-displays/tcltk/stars	SDFTK
instruments/stars	SDFINSTKERNEL

At a minimum, you will need to set the variable corresponding to the 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.

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 and AIX, 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.

Platform-Specific Issues

HP-UX 10.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
```

HP-UX 11.x

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

Calls to casting operators to non-built in types, like `Complex` and `Fix`, don't work in VC++ 5. The compiler thinks that a constructor is being called for the type to be casted, even though one doesn't exist. Calling the casting operator explicitly will work. Code which tries to cast in either of these ways:

```
Complex temp1 = Complex(input1%0);
Fix temp2 = (Fix)input2%0;
```

must be written like this:

```
Complex temp1 = (input1%0).operator Complex();
Fix temp2 = (input2%0).operator Fix();
```

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.

AIX

If you are compiling under AIX, you may get a warning message regarding duplicate symbols. You can safely ignore these messages. These messages are the result of a known bug in the Visual Age C++ compiler. Here are some examples of these warning messages:

- ld: 0711-224 WARNING: Duplicate symbol: __cdtors
- ld: 0711-224 WARNING: Duplicate symbol: .Star::begin()
- ld: 0711-224 WARNING: Duplicate symbol: .DataFlowStar::getDisable()

Chapter 14: Writing Component Models

As described in [Chapter 13, 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 14-21

“Writing Timed Components” on page 14-33

“Using the Agilent Ptolemy Preprocessor Language” on page 14-1

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.xml* extension for auto-documentation generation.

Rectangular Pulse Star Example

To make things clear, let’s start with an example: a rectangular pulse star in the file *SDFRect.pl*. See [Figure 14-1](#). 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:

Writing Component Models

```
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 14-1. 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 14-1 provides an alphabetical list of the items that can appear in a *defstar* directive, including a summary of directives.

Table 14-1. Summary of Items Used to Define a Star

Keyword	Summary	Required	Page
acknowledge	The names of other contributors to the star	No	14-7
attributes	Attributes for the star	No	14-8
attributes	Attributes for PortHoles	No	14-8
attributes	Attributes for the States	No	14-8
author*	The name(s) of the author(s)	No	14-7
begin	C++ code to execute at start time, <i>after</i> the schedule setup	No	14-17
* Indicates a minimum set of the most useful items.			

Table 14-1. Summary of Items Used to Define a Star (continued)

Keyword	Summary	Required	Page
ccinclude	Specify other files to include in the <code>.cc</code> file	No	14-19
code	C++ code to include in the <code>.cc</code> file outside the class definition	No	14-19
conscalls	Define constructor calls for members of the star class	No	14-16
constructor	C++ code to include in the constructor for the star	No	14-16
copyright	Copyright information to include in the generated code	No	14-9
derived	Alternate form of <i>derivedfrom</i>	No	14-6
derivedfrom	The base class, which may also be a star	No	14-6
desc	Alternate form of <i>descriptor</i>	No	14-7
descriptor*	A short summary of the functionality of star	No	14-7
destructor	C++ code to include in the destructor for the star	No	14-17
domain*	The domain and the prefix of the name of a class	Yes	14-6
explanation	Full documentation optionally using <i>troff</i> , <i>eqn</i> , and <i>tbl</i> formats	No	14-10
go*	C++ code to execute when the star fires	No	14-18
header	C++ code to include in the <code>.h</code> file, before the class definition	No	14-19
hinclude	Specify other files to include in the <code>.h</code> file	No	14-19
htmldoc	Full documentation optionally using <i>troff</i> , <i>eqn</i> , and <i>tbl</i> formats	No	14-10
inmulti	Define a set of inputs	No	14-15
input*	Define an input to the star	No	14-15
location	Component library (palette) name where user will find the star	No	14-9
method	Define a member function for the star class	No	14-19
name*	The name of the star and the root of the name of the class	Yes	14-6
outmulti	Define a set of outputs	No	14-15
output*	Define an output from the star	No	14-15
private	Define private data members of the star class	No	14-18
protected	Defined protected data members of the star class	No	14-18
public	Define public data members of the star class	No	14-18
setup*	C++ code to execute at start time, <i>before</i> the scheduler setup	No	14-17
state	Define a state or parameter	No	14-10

* Indicates a minimum set of the most useful items.

Table 14-1. Summary of Items Used to Define a Star (continued)

Keyword	Summary	Required	Page
version	Version number and date	No	14-7
wrapup*	C++ code to invoke at the end of a run (if no error occurred)	No	14-18
vendor	Name of company that authors component. All shipped with Agilent Ptolemy are marked HPEEsof	No	14-20
* 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 14-2, together with subitems of other directives.

Table 14-2. Directive Subitem

Items	Subitems and Descriptions	Required	Page
inmulti, input	<i>name</i> (Name of port or group of ports) <i>type</i> (Data type of input and output particles) <i>descriptor</i> (Summary of function of the input) <i>numtokens</i> (Number of tokens consumed by the port; useful only for dataflow domains)	Yes No No No	14-15
method, virtual method, inline method, pure virtual method, inline virtual method	<i>name</i> (Name of the method) <i>access</i> (Private, protected, or public) <i>arglist</i> (Arguments to the method) <i>type</i> (Return type of the method) <i>code</i> (C++ code defining the method)	Yes No No No If not pure	14-19
outmulti, output	<i>name</i> (Name of port or group of ports) <i>type</i> (Data type of output particles) <i>descriptor</i> (Summary of the functions of output) <i>numtokens</i> (Number of tokens produced by port; useful only for dataflow designs)	Yes No No No	14-15
state	<i>name</i> (Name of the state variable) <i>type</i> (Data type of the state variable) <i>default</i> (Default initial value; always a string) <i>descriptor</i> (Summary of function of state) <i>attributes</i> (State attributes for simulator) <i>units</i> (Type of dimensional units associated with state) <i>enumlist</i> (list of enumeration options) <i>enumlabels</i> (list of alternate names for enumeration options) <i>extensions</i> (list of file extensions for a <i>filename</i> state - default is txt)	Yes Yes No No No No No No No No No	14-10

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.

descriptor

This optional item defines a short description of the class. This description is displayed by the Advanced Design System design environment for this star in the Library list. It has the syntax

```
descriptor {text}
```

where text is simply a section of text that will become the short descriptor of the star. You can also write desc instead of descriptor. A principal use of the short descriptor is to get on-screen help. The following are legal descriptors:

```
desc {A one line descriptor.}
```

or

```
desc {A multi-line descriptor. The same line breaks and spacing will be used when  
the descriptor is displayed on the screen.}
```

By convention, in these descriptors, references to the names of states, inputs, and outputs should be enclosed in quotation marks. If the descriptor seems to get long, augment it with the explanation or `htmldoc` directive, explained below. However, it should be long enough so that it is sufficient to explain the function of the star.

version

This optional item contains entries as follows.

```
version {@/#} $Source: <dir>/my_model.pl $ $Revision: number $ $Date:  
YR/MO/DA $}
```

where the `<dir>` is the source code control directory, the number is a version number, and the YR/MO/DA is the version date.

author

This optional entry identifies the author or authors of the star. Syntax:

```
author {author1, author2 and author3}
```

Any set of characters between the braces will be interpreted as a list of author names.

acknowledge

This optional entry attaches an acknowledgment section to the documentation. Syntax:

acknowledge {arbitrary single line of text}

attributes (for Stars)

This optional entry defines star attributes with syntax:

```
attributes {attribute | attribute | ...}
```

where attributes are separated by the symbol |.

The only possible attribute is:

S_HIDDEN The star is invisible in the design environment. This is typically used for stars that are used only as base stars for other stars.

By default, a star is visible.

attributes (for PortHoles)

This optional entry defines PortHole attributes with syntax:

```
attributes {attribute | attribute | ...}
```

where attributes are separated by the symbol |.

Possible attributes are:

P_HIDDEN The port is invisible in the design environment.

P_VISIBLE The port is visible in the design environment.

P_OPTIONAL The port can be left unconnected. Note, the star code should not read data if the port is not connected. To test whether a port is connected or not, call the function `Port_Name.far()`, where `Port_Name` is the name of the port. If the function returns a NULL pointer, the port is not connected, otherwise it is connected.

P_REQUIRED The port is required to be connected.

By default, all PortHoles are visible and require a connection. Note that the simulator will connect BlackHole models to any unconnected output PortHoles. The BlackHole model itself is hidden.

attributes (for States)

This optional entry defines state attributes with syntax:

```
attributes {attribute | attribute | ...}
```

where attributes are separated by the symbol |.

Possible attributes are:

A_CONSTANT The state is constant during execution of the star.

A_NONCONSTANT The state can change during the execution of the star.

A_SETTABLE The state is visible in the design environment.

A_NONSETTABLE The state is invisible in the design environment.

A_SWEEPABLE The state can be optimized or swept.

A_NONSWEEPABLE The state can neither be optimized nor swept.

A_SCHEMDISPLAY The state is visible and editable on the ADS schematic.

A_NOSCHEMDISPLAY The state is visible and editable only in the Edit Component Parameter dialog box.

By default a State is constant, settable, sweepable, and displayed on the schematic.

copyright

This optional entry attaches a copyright notice to the *.h*, *.cc*, and *.t* files. Syntax:

```
copyright {copyright information}
```

For example, we use the following:

```
copyright {Copyright (c) Agilent Technologies 2000. All rights reserved.}
```

The copyright can span multiple lines, just like a descriptor.

location

This optional item is the name of the schematic library in which the user will find the star. Syntax:

```
location {<main libraryname,>libraryname}
```

where *libraryname* is the location of the star in the Advanced Design System design environment Signal Processing schematic and 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  
}
```

htmldoc

This optional item is used to give a longer explanation of the star's function. Syntax:

```
explanation {  
  body  
}
```

state

This optional item is used to define a state or parameter. The following is an example of a state definition:

```
state {  
  name {gain}  
  type {int}  
  default {1.0}  
  units {UNITLESS_UNIT}  
  desc {output gain}  
  attributes {A_CONSTANT | A_SETTABLE}  
}
```

The following ten types of subitems may appear in a state definition, in any order: *name*, *type*, *default*, *desc*, *units*, *enum*, *enumlist*, *enumlabel*, *extensions*, *attributes*.

- The *name* field (required) is the name of the state.
- The *type* field (required) is its type, which may be one of *int*, *float*, *string*, *complex*, *fix*, *intarray*, *floatarray*, *complexarray*, *precision*, *stringarray*, *filename*, *enum*, *query*, or *boolean*. Case is ignored for the type argument.

- The *default* field (optional) specifies the default initial value of the state. Its argument is either a string (enclosed in quotation marks) or a numeric value. The preceding entry could equivalently have been written:

```
default { "1.0" }
```

Furthermore, if a particularly long default is required, as for example when initializing an array, the string can be broken into a sequence of strings. The following example shows the default for a *ComplexArray*.

```
default {  
"(-.040609,0.0) (-.001628,0.0) (.17853,0.0) (.37665,0.0)"  
"(.37665,0.0) (.17853,0.0) (-.001628,0.0) (-.040609,0.0)"  
}
```

For complex states, the syntax for the default value is *(real, imag)* where *real* and *imag* evaluate to integers or floats.

The *precision* state is used to give the precision of fixed-point values. These values may be other states or may be internal to the star. The default can be specified in either of two ways:

Method 1: As a string like “3.2”, or more generally “*m.n*”, where *m* is the number of integer bits (to the left of the binary point) and *n* is the number of fractional bits (to the right of the binary point). Thus length is *m+n*.

Method 2: A string like “24/32” which means 24 fraction bits from a total length of 32. This format is often more convenient because the word length often remains constant while the number of fraction bits changes with the normalization being used.

In both cases, the sign bit counts as one of the integer bits, so this number must be at least one.

For *enum* states, the default value may only be one of the values listed in the *enumlist* field, in quotes (“value”).

For *filename* states, the default value is the name of a file.

Units, such as MHz, msec, etc., can also be used in the default field, e.g., {“3 msec”}. However, this is not recommended since the value will not be displayed properly on the Schematic when the user changes the Units Scale Factors from the Options > Preferences > Units/Scale tab.

Expressions can also be used to set the default value. For example, the default field of a float state can be {"sin(0.3)} or the default field of a complex state can be {"polar(3.2, 1.13)}.

An example of the default field for each type is shown in [Table 14-3](#).

- 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 14-3. Default Fields

Type	Example Default
int	default {3}
float	default {3}
fix	default {1.25}
complex	default {"(1.25, 2.5)"}
string	default {"string value"}
precision	default {"24/32"} or equivalently default {"8.24"}
enum	default {"value 1"}
filename	default {"user/abc/xyz/lmn.txt"}
intarray	default {1, 2, 4, 7, 9}
floatarray	default {1.25, 3.50, 6.75}
complexarray	default {"(1.25, 2.5) (2.4, -2.3) (-1.2, -2.2)"}
stringarray	default {"Button1" "Button 2"}

States with the *A_NONCONSTANT* attribute may change when the star is run. If a state has the *A_SETTABLE* attribute, then user interfaces will enable the user to enter values for this state. States without this attribute are not

presented to the user; such states always start with their default values as the initial value.

If a state has the `A_SWEEPABLE` attribute, its value can be swept and/or optimized using the appropriate simulation controllers. On the other hand, the `A_NONSWEEPABLE` attribute does not allow sweeping and/or optimizing the state's value.

The `A_SCHEMDISPLAY` attribute is only used for states that are `A_SETTABLE`. If a state has the `A_SCHEMDISPLAY` and `A_SETTABLE` attributes set, the state will be shown on the ADS schematic. If a state has the `A_NOSCHEMDISPLAY` and `A_SETTABLE` attributes set, the state will only be shown in the Edit Component Parameter dialog box.

Note that of all the attributes only `A_SCHEMDISPLAY` and `A_NOSCHEMDISPLAY` can be modified by the user for a specific instance of a component. This is done in the Edit Component Parameters dialog box by setting the “display parameter on schematic” flag appropriately.

If no attributes are specified, the default is `A_CONSTANT|A_SETTABLE|A_SCHEMDISPLAY|A_SWEEPABLE`. Thus, in the above example, the *attributes* directive is unnecessary.

- The *units* field (optional) identifies the set of dimensional scale factors to be associated with this state at the schematic level in Advanced Design System. By default, the value of this field is `UNITLESS_UNIT`, which results in no scale factor association at the schematic level. Other unit options are shown in [Table 14-4](#).

Table 14-4. Unit Options

Option	Function
<code>FREQUENCY_UNIT</code>	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.
<code>TIME_UNIT</code>	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.
<code>ANGLE_UNIT</code>	Results in use of degree angle units at the schematic level. The state in the code will receive a value in terms of degrees.
<code>POWER_UNIT</code>	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.

Table 14-4. Unit Options (continued)

Option	Function
DISTANCE_UNIT	Results in use of distance unit scale factors at the schematic level (m, km, mile, etc.). The state in the code will reveal a value in terms of m.
LENGTH_UNIT	Results in use of length scale and conversion factors at the schematic level (m, mm, cm, in, ft, etc.). The state in the code will receive a value in terms of m.
RESISTANCE_UNIT	Results in use of resistance scale factors at the schematic level (Ohms, KOhms, MOhms, etc.). The state in the code will receive a value in terms of Ohms.
CAPACITANCE_UNIT	Results in use of capacitance scale factors at the schematic level (F, uF, pF, etc.). The state in the code will receive a value in terms of F.
INDUCTANCE_UNIT	Results in use of inductance scale factors at the schematic level (H, mH, uH, etc.). The state in the code will receive a value in terms of H.

- The *enumlist* field is required when the state type is *enum*. The *enumlist* field is a comma separated list of strings.

```
enumlist {value 1, value 2, value 3}
```

Quotes around strings are optional. Spaces and other non-alphanumeric characters can be used. However, when referencing an *enum* value in the code for the star, all non-alphanumeric characters must be replaced with an underbar (`_`). For example, “value 1” should be referenced as “value_1”.

- The *enumlabels* field is optional and available for use only when the state type is *enum*. The *enumlabels* field contains name abbreviations for each *enumlist* value. The alternate names are for use only at the schematic level in Advanced Design System as a short mnemonic for the full enumeration value.

```
enumlabels {v1, v2, v3}
```

The label *v1* is used only as an abbreviation for *value 1*.

Two very commonly used enums are predefined: query with `enumlist {NO, YES}` and boolean with `enumlist {FALSE, TRUE}`. Below is an example showing their use:

```
state {
    name {periodic}
```

```

    type {query}
    default {YES}
}

```

- 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 14-28 and “Array States” on page 14-30.

input, output, inmulti, outmulti

These optional items are used to define a porthole, which may be an input, output porthole or an input, output multiporthole. Bidirectional ports are not supported. Like *state*, it contains subitems. The following is an example:

```

input {
    name {signalIn}
    type {complex}
    numtokens {2}
    desc {A complex input that consumes 2 input particles.}
}

```

Here, *name* specifies the porthole name. This is a required item.

The keyword *type* specifies the particle type. The scalar types are *int*, *float*, *fix*, *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 15, 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:

```
type {= name }
```

where *name* is the name of another porthole. This indicates that this porthole inherits its type from the specified porthole. For example, here is a portion of the definition of the SDF *Fork* star:

```
input {
    name{input}
    type{ANYTYPE}
}
outmulti {
    name{output}
    type{= input}
    desc{type is inherited from the input}
}
```

constructor

This optional item enables the user to specify extra C++ code to be executed in the constructor for the class. This code will be executed *after* any automatically generated code in the constructor that initializes portholes, states, etc. The syntax is:

```
constructor {body}
```

where *body* is a piece of C++ code. It can be of any length. Note that the constructor is invoked only when the class is first instantiated; actions that must be performed before every simulation run should appear in the *setup* or *begin* methods, not the constructor.

conscalls

With this optional item, you might have data members in your star that have constructors requiring arguments. These members would be added by using the *public*, *private*, or *protected* keywords. If you have such members, the *conscalls* keyword provides a mechanism for passing arguments to the constructors of those

members. Simply list the names of the members followed by the list of constructor arguments for each, separated by commas if there is more than one. The syntax is:

```
conscalls {member1(arglist), member2(arglist)}
```

Note that *member1*, and *member2* should have been previously defined in a *public*, *private*, or *protected* section. (See the subsequent descriptions of these keywords.)

destructor

This optional item inserts code into the destructor for the class. The syntax is:

```
destructor {body}
```

You generally need a destructor only if you allocate memory in the constructor, *begin* method, or *setup* method; termination functions that happen with every run should appear in the *wrapup* function. (*Wrapup* is not called if an error occurs. See subsequent description of the *wrapup* keyword.) The optional keyword *inline* may appear before *destructor*. If so, the destructor function definition appears inline, in the header file. Since the destructor for all stars is virtual, this is only a win when the star is used as a base for derivation.

setup

This optional item defines the *setup* method, which is called every time the simulation is started, *before* any compile-time scheduling is performed. The syntax is:

```
setup {body}
```

The optional keyword *inline* may appear before the *setup* keyword. It is common for this method to set parameters of input and output portholes, and to initialize states. For an explanation of the code syntax for doing this, refer to the section, [“Reading Inputs and Writing Outputs” on page 14-24](#). In some domains, with some targets, the *setup* method may be called more than once during initiation. You must keep this in mind if you use it to allocate or initialize memory.

begin

This optional item defines the *begin* method, which is called every time the simulation is started, but *after* the scheduler *setup* method is called (that is, after any compile-time scheduling is performed). The syntax is:

```
begin {body}
```

This method can be used to allocate and initialize memory. It is especially useful when data structures are shared across multiple instances of a star. It is always called exactly once when a simulation is started.

go

This optional item defines the action taken by the star when it is fired. The syntax is:

```
go {body}
```

The optional keyword *inline* may appear before the *go* keyword. The *go* method will typically read input particles and write outputs, and will be invoked many times during the course of a simulation. For an explanation of the code syntax for the body, refer to the section, [“Reading Inputs and Writing Outputs” on page 14-24](#).

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 14-24](#). 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:

```
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.

Warning Messages

Warnings have their messages automatically prefixed with the token *Warning:*, and allow the simulation to conclude. Warning messages are typically used in the *setup()* and *go()* methods. They can be created using the *Error::warn()* methods. These methods have the following prototypes:

```
Error::warn(const char *, const char * = 0, const char * = 0);
```

```
Error::warn(const NamedObj&, const char *, const char * = 0, const char * = 0);
```

Where possible, the second method should be used so that the name of the *NamedObj* can be displayed along with the message. The named object can be **this* to mean the current star instance.

Error Messages

Errors have their messages automatically prefixed with the token *ERROR*.; and result in stopping the simulation. Error messages used before the *go()* method are called during a simulation and should not cause the simulation to stop until after the completion of the simulation initialization and setup.

During the *setup()* method, state values should be checked for any value range error. If an error exists in the state value, the *State::rangeError()* method should be used:

```
<state_name>.rangeError(const char *);
```

where the *const char ** is a string that defines the required state range.

Examples:

```
FCarrier.rangeError(">= 0.0");
```

```
Top.rangeError("> Bottom");
```

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

```
Error::initialization(const char *, const char * = 0, const char * = 0);
```

```
Error::initialization(const NamedObj&, const char *, const char * = 0, const char * = 0);
```

Where possible, the second method should be used so that the name of the *NamedObj* can be displayed along with the message. The named object can be **this* to mean the current star instance.

Error messages used in the *go()* method will not cause the simulation to stop until after the current *go()* method is complete.

These error messages can be created using the *Error::abortRun()* methods. These methods have the following prototypes:

```
Error::abortRun(const char *, const char * = 0, const char * = 0);
```

```
Error::abortRun(const NamedObj&, const char *, const char * = 0, const char * = 0);
```

Where possible, the second method should be used so that the name of the *NamedObj* can be displayed along with the message. The named object can be **this* to mean the current star instance.

Reading Inputs and Writing Outputs

The precise mechanism for references to input and output portholes depends somewhat on the domain. This is because stars in the domain *XXX* use objects of class *InXXXPort* and *OutXXXPort* (derived from *PortHole*) for input and output, respectively. The examples used here are for the SDF (or TSDF) domain. See the appropriate domain 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,

```
input {
    name {in}
    type {float}
}
```

a member named *in*, of type *InSDFPort* (*InTSDFPort*), will become part of the star.

We are not usually interested in directly accessing these porthole classes, but rather wish to read or write data through the portholes. All data passing through a porthole is derived from base class *Particle*. Each particle contains data of the type specified in the *type* subdirective of the *input* or *output* directive.

The operator `%` operating on a porthole returns a reference to a particle. Consider the following example:

```
go {
    Particle& currentSample = in%0;
    Particle& pastSample = in%1;
    ...
}
```

The right-hand argument to the `%` operator specifies the delay of the access. A zero always means the most recent particle. A one means the particle arriving just before the most recent particle. This also applies to outputs. Given an output named *out*, the particles that are read from *in* can be written to *out* in the same order as follows:

```
go {
    ...
    out%1 = pastSample;
```

```
    out%0 = currentSample;
}
```

This works because *out%n* returns a *reference* to a particle, and hence can accept an assignment. The assignment operator for the class *Particle* is overloaded to make a copy of the data field of the particle.

Operating directly on class *Particle*, as in the above examples, is useful for writing stars that accept *anytype* of input. The operations don't need to concern themselves with the type of data contained by the particle. But it is far more common to operate numerically on the data carried by a particle. This can be done using a cast to a compatible type. For example, since *in* above is of type *float*, its data can be accessed as follows:

```
go {
    Particle& currentSample = in%0;
    double value = double(currentSample);
    ...
}
```

or more concisely,

```
go {
    double value = double(in%0);
    ...
}
```

The expression *double(in%0)* can be used anywhere that a double can be used. In many contexts, where there is no ambiguity, the conversion operator can be omitted:

```
double value = in%0;
```

However, since conversion operators are defined to convert particles to several types, it is often necessary to indicate precisely which type conversion is desired.

To write data to an output porthole, note that the right-hand side of the assignment operator should be of type *Particle*, as shown in the above example. An operator *<<* is defined for particle classes to make this more convenient. Consider the following example:

```
go {
    float t;
    t = some value to be sent to the output
    out%0 << t;
}
```

Note the distinction between the `<<` operator and the assignment operator. The latter operator copies *Particles*, the former operator loads data into particles. The type of the right-side operand of `<<` may be *int*, *float*, *double*, *Fix*, *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 15, 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:

```
setup {
    name.setSDFParams(multiplicity, past)
}
```

where *name* is the name of the input or output porthole, *multiplicity* is the number of particles consumed or produced, and *past* is the maximum value that *offset* can take in any expression of the form *name%offset*. For example, if the *go* method references *name%0* and *name%1*, then *past* would have to be at least one. It is zero by default.

Multiple PortHoles

Sometimes a star should be defined with *n* input portholes or *n* output portholes, where *n* is variable. This is supported by the class *MultiPortHole*, and its derived classes. An object of this class has a sequential list of *PortHoles*. For *SDF (TSDF)*, we have the specialized derived class *MultiInSDFPort (MultiInTSDFPort)*, which contains *InSDFPorts (InTSDFPorts)* and *MultiOutSDFPort (MultiOutTSDFPort)*, which contains *OutSDFPorts (OutTSDFPorts)*.

Defining a multiple porthole is easy, as illustrated below:

```
defstar {
    ...
    inmulti {
        name {input_name}
        type {input_type}
    }
}
```



```

    outmulti {
        name {output_name}
        type {output_type}
    }
    ...
}

```

To successively access individual portholes in a *MultiPortHole*, the *MPHIter* iterator class should be used. Consider the following code segment from the definition of the *SDF Fork (TSDF Fork)* star:

```

input {
    name{input}
    type{ANYTYPE}
}
outmulti {
    name{output}
    type{= input}
}
go {
    MPHIter nextp(output);
    PortHole* p;
    while ((p = nextp++) != 0)
        (*p)%0 = input%0;
}

```

A single input porthole supplies a particle that gets copied to any number of output portholes. The *type* of the output *MultiPortHole* is inherited from the type of the input. The first line of the *go* method creates an *MPHIter* iterator called *nextp*, initialized to point to portholes in *output*. The ++ operator on the iterator returns a pointer to the next porthole in the list, until there are no more portholes, at which time it returns *NULL*. So the *while* construct steps through all output portholes, copying the input particle data to each one.

Consider another example, taken from the *SDF Add* star:

```

inmulti {
    name {input}
    type {float}
}
output {
    name {output}
}

```

```

    type {float}
  }
  go {
    MPHIter nexti(input);
    PortHole *p;
    double sum = 0.0;
    while ((p = nexti++) != 0)
      sum += double>(*p)%0);
    output%0 << sum;
  }

```

Again, an *MPHIter* iterator named *nexti* is created and used to access the inputs.

The *numberPorts* method of class *MultiPortHole*, which returns the number of ports, is occasionally useful. This is called simply as *portname.numberPorts()*, and it returns an *int*.

Type Conversion

The type conversion operators and << operators are defined as virtual methods in the base class *Particle*. There are never really objects of class *Particle* in the system. Instead, there are objects of class *IntParticle*, *FloatParticle*, *ComplexParticle*, *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 *anytype*.

States

A state is defined by the *state* directive. The star can use a state to store data values, remembering them from one invocation to another. States differ from ordinary members of the star (defined by the *public*, *protected*, and *private* directives) in that they have a name, and can be accessed from outside the star in systematic ways. For instance, the Advanced Design System design environment enables you to set any

state with the *A_SETTABLE* attribute to some value prior to a run; this is done via the on schematic value entry or the Edit Component dialog. The state attributes are set in the *state* directive.

A state may be modified by the star code during a run. To mark a state as one that gets modified during a run, use the attribute *A_NONCONSTANT*. There is currently no mechanism for checking the correctness of these attributes.

All states are derived from the base class *State*, defined in the 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:

```
state {
  name {myState}
  type {float}
  default {1.0}
  descriptor {Gain parameter.}
}
```

This will define a member of class *FloatState* with default value 1.0. No attributes are defined, so *A_CONSTANT* and *A_SETTABLE*, the default attributes, are assumed. To use the value of a state, it should be cast to type *double*, either explicitly by the programmer or implicitly by the context. For example, the value of this state can be accessed in the *go* method as follows:

```
go {
  output%0 << double(myState) * double(input%0);
}
```

The references to *input* and *output* are explained above. The reference to *myState* has an explicit cast to *double*; this cast is defined in the *FloatState* class. Similarly, a cast to *int* is available for *IntState*, to *Complex* for *ComplexState*, and to *const char** for *Stringstate*). In principle, it is possible to rely on the compiler to automatically invoke this cast. However, note the following warning.

Explicit casting should be used whenever a state is used in an expression. For example, from the setup method of the *SDFChop* star, in which *use_past_inputs* is an integer state,

```

if (int(use_past_inputs))
    input.setSDFParams(int(nread),int(nread)+int(offset)-1);
else
    input.setSDFParams(int(nread),int(nread)-1);

```

Note that the type *Complex* is not a fundamental part of C++. We have implemented a subset of the *Complex* class as defined by several library vendors. We use our own version for maximum portability. Using the *ComplexState* class automatically ensures the inclusion of the appropriate header files. A member of the *Complex* class can be initialized and operated upon any number of ways. For details, refer to the section, “The Complex Data Type,” in Chapter 12, “Using Data Types in Model Building.”

A state may be updated by ordinary assignment in C++, as in the following lines:

```

double t = expression;
myState = t;

```

This works because the *FloatState* class definition has overloaded the assignment operator (=) to set its value from a *double*. Similarly, an *IntState* can be set from an *int*, and a *StringState* can be set from a *char** or *const char**.

Array States

The *ArrayState* classes (*FloatArrayState*, *IntArrayState* and *ComplexArrayState*) are used to store data arrays. For example,

```

state {
    name {taps}
    type {FloatArray}
    default {"0.0 0.0 0.0 0.0"}
    descriptor {An array of length four.}
}

```

defines an array of type *double* with dimension four, with each element initialized to zero. Quotes must surround the initial values. Alternatively, you can specify a file name with the prefix <. If you have a file named *foo* that contains the default values for an array state, you can write:

```

default {"< foo"}

```

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

```
default {"< ~/user_name/directory/foo"}
```

The format of the file is also a sequence of data separated by spaces (or new lines, tabs, or commas). File input can be combined with direct data input as in:

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

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

```
default {"1.0 [5]"}
default {"1.0 1.0 1.0 1.0 1.0"}
```

are equivalent. Any integer expression may appear inside the brackets `[]`. The number of elements in an *ArrayState* can be determined by calling its *size* method. The size is not specified explicitly, but is calculated by scanning the default value.

As an example of how to access the elements of an *ArrayState*, suppose *fState* is a *FloatState* and *aState* is a *FloatArrayState*. The access points, like those in the following lines, are routine:

```
fState = aState[1] + 0.5;
aState[1] = (double)fState * 10.0;
aState[0] = (double)fState * aState[2];
```

For a more complete example of the use of *FloatArrayState*, consider the *FIR* star defined below. Note that this is a simplified version of the SDF *FIR* star and does not permit interpolation or decimation.

```
defstar {
  name {FIR}
  domain {SDF}
  desc {
    A Finite Impulse Response (FIR) filter.
  }
  input {
    name {signalIn}
    type {float}
  }
  output {
    name {signalOut}
    type {float}
  }
  state {
```

```

name {taps}
type {floatarray}
default { "-.04 -.001 .17 .37 .37 .17 -.0018 -.04" }
desc {Filter tap values.}
}
setup {
// tell the PortHole the maximum delay we will use
signalIn.setSDFParams(1, taps.size() - 1);
}
go {
double out = 0.0;
for (int i = 0; i < taps.size(); i++)
out += taps[i] * double(signalIn%i);
signalOut%0 << out;
}
}

```

Notice the *setup* method; this is necessary to allocate a buffer in the input *PortHole* large enough to hold the particles that are accessed in the *go* method. Notice also the use of the *size* method of the *FloatArrayState*.

Modifying PortHoles and States in Derived Classes

When one star is derived from another, it inherits all the states of the base class star. Sometimes we want to modify some aspect of the behavior of a base class state in the derived class. This is done by placing calls to member functions of the state in the constructor of the derived star. Useful functions include *setInitValue* to change the default value, and *setAttributes* and *clearAttributes* to modify attributes.

When creating new stars derived from stars already in the system, you will often also wish to customize them by adding new ports or states. In addition, you may wish to remove ports or states. Although, strictly speaking, you cannot do this, you can achieve the desired effect by simply hiding them from the user.

The following code will hide a particular state named *statename* from the user:

```

constructor {
    statename.clearAttributes(A_SETTABLE);
}

```

Thus, when the user observes the available states for this star in the Advanced Design System design environment, *statename* will not appear as one of the star

parameters. Of course, the state can still be set and used within the code defining the star.

The same effect can be achieved with outputs or inputs. For instance, given an output named *output*, you can use the following code:

```
constructor {  
    output.setAttributes(P_HIDDEN);  
}
```

This means that when you create an icon for this star, no terminal appears for this port. This is most useful when *output* is a multiporthole, because there will then be zero instances of the individual portholes.

This technique can also be used to hide individual portholes. However, it must be used with caution because the porthole still remain. Most domains do not allow disconnected portholes, and will flag an error. You can explicitly connect the port within the body of the star.

Writing Timed Components

Writing Timed components using hpeesoflang is almost identical to writing any other star. Following are the primary points of distinction:

- Receiving Timed data

To receive the data field of the Timed data via input TSDFPortHole, use the following method:

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

where *n* is the current value of the input stream. For example, if the Timed input port is named *in*, then

```
in.getIQData(0)
```

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

Similarly, the methods

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

and

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

return the Flavor and Fc associated with incoming Timed particle.

- Sending data

As described in the preceding section “[Reading Inputs and Writing Outputs](#)” on [page 14-24](#), the operator << is used to load the output port with Timed data. For example, given the Timed ports *out1*, *out2*, the following will output Baseband and ComplexEnv flavor Timed data at *out1* and *out2* ports. Note that the other attributes of Timed particle are set by the engine.

```

go{
  double x;
  Complex z;
  .....
  out1%0 << x;
  out2%0 << z;
}

```

- Fc propagation

When a TSDF star is *changing (or re-setting)* the carrier frequency *Fc*, a TSDFStar method should be used in the star setup as follows:

```
TSDFStar::propagateFc(double fc)
```

If this method is not explicitly used, the virtual method is used, which sets the output carrier frequency equal to the maximum input carrier frequency.

Example of Writing Timed Components

```

method {
  name {propagateFc}
  access {protected}
  arglist {"(double *fcin)"}
  type {void}
  code {
    output.setCarrierFrequency(dummy);
  }
}

```

- TStep propagation

When a TSDF star is *changing or re-setting* the TStep (for example a source), a TSDFStar method should be used in the star setup, as follows:

```
TSDFPortHole::setTimeStep(double timestep)
```


- ComplexToTimed Converter example

```
defstar {
  name {CxToTimed}
  domain {TSDF}
  desc {Converts a Complex signal to Timed. Given the
  complex number (a+bj) at input, the output is a
  ComplexEnv Timed signal
  {(I + jQ),fc} where I=a, Q=b and fc is a parameter.}
  copyright {Copyright (c) Hewlett-Packard Company 1997}
  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);
  }
}
```

```

// 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}
  }
}

```

```

    desc {Increment from one sample to the next.}
}
state {
    name {value}
    type {float}
    default {0.0}
    desc {Initial (or latest) value output by Ramp.}
    attributes {A_SETTABLE|A_NONCONSTANT}
}
go {
    double t = double(value);
    output%0 << t;
    t += step;
    value = t;
}
}

```

The next example is the *Gain* star, which multiplies its input by a constant and outputs the result:

```

defstar {
    name { Gain}
    domain {SDF}
    desc {Amplifier: output is input times "gain" (default 1.0).}
    input {
        name {input}
        type {float}
    }
    output {
        name {output}
        type {float}
    }
    state {
        name {gain}
        type {float}
        default {"1.0"}
        desc {Gain of the star.}
    }
    go {
        output%0 << double(gain) * double(input%0);
    }
}

```

```

    }
}

```

The following example of the *Printer* star illustrates multiple inputs, *ANYTYPE* inputs, and the use of the *print* method of the *Particle* class.

```

defstar {
  name {Printer}
  domain {SDF}
  inmulti {
    name {input}
    type {ANYTYPE}
  }
  state {
    name {fileName}
    type {string}
    default {"<cout>"}
    desc {Filename for output.}
  }
  hinclude {"pt_fstream.h"}
  protected {
    pt_ofstream *p_out;
  }
  constructor {p_out = 0;}
  destructor {LOG_DEL; delete p_out;}
  setup {
    delete p_out;
    p_out = new pt_ofstream(fileName);
  }
  go {
    pt_ofstream& output = *p_out;
    MPHIter nexti(input);
    PortHole* p;
    while ((p = nexti++) != 0)
      output << ((*p)%0).print() << "\t";
    output << "\n";
  }
}

```

This star is *polymorphic* since it can operate on any type of input. Note that the default value of the output filename is `<cout>`, which causes the output to go to the standard output.

Preventing Memory Leaks in C++ Code

Memory leaks occur when new memory is allocated dynamically and never deallocated. In C programs, new memory is allocated by the *malloc* or *calloc* functions, and deallocated by the *free* function. In C++, new memory is usually allocated by the *new* operator and deallocated by the *delete* or the *delete []* operator. The problem with memory leaks is that they accumulate over time and, if left unchecked, may cripple or even crash a program. Agilent EEsof has taken extensive steps to eliminate memory leaks in the 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[bufflen];
```

This code should be

```
delete [] thestring;  
thestring = new char[bufflen];
```

Using *delete* is not necessary in a class' constructor because the data member would not have been previously allocated.

- 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:

```
StringList sl = msg;
sl << file << ": " << sys_errlist[errno];
ErrAdd(sl);
```

The *errAdd* function takes a *const char** argument, so *sl* is automatically converted to a *const char** string by the C++ compiler.

Instead of using the new and delete operators, it is tempting to use constructs like:

```
char localstring[buflen + 1];
```

in which *buflen* is a variable. This is because the compiler will then automatically handle memory deallocation. Unfortunately, this syntax is a Gnu extension and is not portable to other C++ compilers. Instead, the *StringList* and *InfString* classes should be used, as in the previous example involving *localstring*.

Sometimes the return value from a routine that returns dynamic memory is not stored and, therefore, the pointer to the dynamic memory gets lost. This occurs, for example, in nested function calls. Code such as

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

should instead be written as

```
const char* newstring = savestring(s);
puts(newstring);
delete [] newstring;
```

Several features in 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:

Writing Component Models

```
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 $
$Revision: 1.0 $ $Date: 1997/10/28 16:26:58 $}
// This version field to be changed as needed for HMS source code control by the user
author {Author's name}
acknowledge {arbitrary single line of text to acknowledge others}
location {my_model_library_location} // Name of Library used in the Schematic
attributes {S_USER} // or S_HIDDEN
derivedfrom {base_star_name}        // Optional: delete if not used
copyright {
Copyright (c) Hewlett-Packard Company 1997
All rights reserved.
}
explanation {my_model_explanation}
// Use as many lines as needed to describe the star, it's purpose,
// algorithm, application, references, or other information to
// document this component
// Define a defstate for each parameter
defstate {
name {my_state_name}    // Limit length to one line 30 characters maximum
                        // No spaces. Only alpha-numeric characters and underbar.
                        // Name should be constructed as one or more word segments
                        // with each word segment beginning with a capital letter.
type {my_state_type} // Options: int, fix, float, complex, string, precision,
                        // intarray, fixarray, floatarray, complexarray, stringarray
                        // For the enum state, see the next defstate{} example
// For the filename state, see the following defstate{} example
default {my_state_default_value}
// Example int:          1
// Example float:       1.25
// Example fix:         1.25
// Example complex:     "(1.25, 2.5)"
// Example string:      "my string"
// Example precision:   2.14
// Example intarray:    1, 2, 3, 6, 9
// Example float array: 1.25, 3.50, 6.75
// Example complexarray: "(1.25, 2.5) (2.4, -2.3) (-1.2, -2.2)"
// Example stringarray: "Button 1" "Button 2"
units {UNITLESS_UNIT}
// Options: STRING_UNIT, UNITLESS_UNIT,
// FREQUENCY_UNIT, TIME_UNIT, ANGLE_UNIT
// Note: ANGLE_UNIT for phase in degrees
// Other units are available for resistance, length, etc., but might
```



```

        // not be relevant to numeric stars
desc {my_state_description}
    // Begin with a short phrase defining this state and ending with
    // a period and a space. This initial sentence will be used in the
    // AEL for this star. This initial sentence may be followed
    // with additional content to describe this state and its use
attributes {A_SETTABLE | A_NONCONSTANT}
    // These attributes are for states the for use at the schematic
    // level. If a state is to be hidden from the schematic, it can be
    // listed as A_NONSETTABLE | A_NONCONSTANT
}
// Define a defstate for each parameter; example for enumerated state
defstate {
    name {my_state_name} // Limit length to one line 30 characters maximum
                        // No spaces. Only alpha-numeric characters and underbar.
                        // Name should be constructed as one or more word segments
                        // with each word segment beginning with a capital letter.

    type {enum} // enumerated state
    default {"option1"} // default in quotes
    desc {my_state_description}
        // Same notes as for the state description apply
    enumlist { option 1, option 2}
        // enumerated list separate with commas
        // each enumeration may contain spaces, underbar or other
        // alpha-numeric characters, but none other
        // Code may be reference the enumeration by use of the option
        // with spaces replaced by underbars.
        // Example: if (my_enum_name == option_1) {
        //     ... code here ...
        // }
    enumlabels { opt 1, opt 2}
        // an abbreviation of the enumlist options, used during AEL
        // generation
    attributes {A_SETTABLE|A_NONCONSTANT}
        // These attributes are for states the for use at the schematic
        // level. If a state is to be hidden from the schematic, it can be
        // listed as A_NONSETTABLE | A_NONCONSTANT
}
// Define a defstate for each parameter; example for file name state
defstate {
    name {my_state_name} // Limit length to one line 30 characters maximum
                        // No spaces. Only alpha-numeric characters and underbar.
                        // Name should be constructed as one or more word segments
                        // with each word segment beginning with a capital letter.

    type {filename} // file name state
    default {"xyz.ext1"} // default in quotes
    desc {my_state_description}
        // Same notes as for the state description apply
    extensions { ext1, ext2, ext3 }
        // extension list separate with commas
        // each extension may contain underbar or other
        // alpha-numeric characters, but none other
    attributes {A_SETTABLE|A_NONCONSTANT}
        // These attributes are for states the for use at the schematic
        // level. If a state is to be hidden from the schematic, it can be
        // listed as A_NONSETTABLE | A_NONCONSTANT
}

```

Writing Component Models

```
}
port_type {      // Options:  input, output, inmulti, outmulti
                // See programmers manual for the use of each port_type
    name {port_name}
    type {port_type}
// When Domain == SDF:
//Options: int, float, fix, complex, message, int_matrix_env,
        // float_matrix_env, complex_matrix_env, fix_matrix_env,
        // anytype
// When Domain == TSDF:
// Options: timed
    desc {port_desc}
}
hinclude {
// Optional: delete if not used
// User specifies other files to include in the .h file
}
header {
// Optional: delete if not used
// User places C/C++ code to include in the .h file, before the class definition
}
ccinclude {
// Optional: delete if not used
// User inserts .cc include files here in quotes with comma separators. Example:
// "file1.cc","file2.cc","file3.cc"
}
private {
// Optional: delete if not used
// Define private data members of the star class
}
protected {
// Optional: delete if not used
// Define protected data members of the star class
}
constructor {
// Optional: delete if not used
// Called when instance created.
// Allows user to specify extra C/C++ code to be executed in the constructor
// for the class.
// This field can initialize the public data member that indicates delays associated
// with input pins
}
conscalls {
// Optional: delete if not used
// Used when data members have constructors and require arguments.
// These members would be added by using the public, private, or
// protected keywords. If such members exist, conscalls provides
// the user with a mechanism for passing arguments to the
// constructors of those members. Example:
// member1(arglist), member2(arglist)
}
setup {
// Optional: delete if not used
// C/C++ code to execute at start time, before the scheduler setup.
// Check each state value for validity
// Example: if (double(state_name) < 0.5) {
```

```

//                               state_name.rangeError(">= 0.5");
//                               }
// See also messaging guidelines for status messages, warning messages,
// error messages
// }
begin {
// Optional: delete if not used
// C/C++ code to execute at start time, after the scheduler setup.
// }
go {
// User supplied C/C++ code here
// }
wrapup {
// Optional: delete if not used
// C/C++ code to invoke at the end of a run (if no error occurred)
// }
destructor {
// Optional: delete if not used
// User C/C++ code to include in the destructor for the star
// }
method {
// Optional: delete if not used
// Define a member function for the star class
// Can also substitute for method:
// virtual method, inline method, pure method, pure virtual method,
// inline virtual method
// name {user defined name}
// access {either private, protected, or public}
// arglist {"(arguments in quotes)"}
// type {the return type of the method}
// code {C/C++ code defining the method}
// }
code {
// Optional: delete if not used
// C/C++ code to include in the .cc file outside the class definition
// }
}
}

```

Writing Sink Models

Sinks are models with inputs but no outputs. The main use of sinks is to write data to files (ASCII, dataset, etc.). The data written to the file could be the raw data collected from the sink's inputs, or the sink's collected data can be processed and then written to the file. The processing of the data can be done during the simulation (in the go method of the sink) or after the simulation has finished (in the wrapup method of the sink).

In order to write a sink model, you first need to understand the concept of a task. In addition, you need to learn how to use tasks and how to write data to a dataset. The following sections describe these concepts and give simple examples showing how they are used.

Understanding Tasks

A task is something that needs to be completed before the simulation can finish. In other words, you can think of a task as something that controls the simulation by keeping it running or by causing it to terminate. The simulator keeps a list of all the tasks that have not completed, called the `TaskList`, and as long as there are tasks in this list it will continue to run the simulation. The simulation terminates as soon as the `TaskList` becomes empty.

Any model can add/remove tasks to/from the `TaskList` at any time during the simulation. However, tasks are particularly useful when used in source or sink models. All 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 14-52](#) is another example of such a sink.

Sink Coding Methodology

Every sink needs to define an object of the class `SinkControl` in the private, protected, or public section of its `.pl` file. This will require that the `TargetTask.h` header file is listed in the `hinclude` section of the `.pl` file and that the variable `KERNEL` (or another

variable that automatically sets `KERNEL` to 1, such as `SDFKERNEL`, `TSDFKERNEL`) is set to 1 in the corresponding make-defs file. The *SinkControl* object should call its *initialize* function in the begin section. The *go* section should look like:

```
go {  
  if (sinkControl.collectData()) {  
    // all the sink go code should be entered her.  
  }  
}
```

where `sinkControl` is an object of type *SinkControl*.

Let's look into what all the above means in more detail. A *SinkControl* object is an object that can add/remove a single task to/from the `TaskList`. In addition, it has an internal timer/counter to keep track of some notion of the simulation time. A *SinkControl* object needs to be initialized before it can be used. This is done by calling its *initialize* function. There are two overloaded versions of this function:

- i) *initialize*(`Block& master`, `double start_value`, `double stop_value`, `double step_value`)
- ii) *initialize*(`Block& master`, `double start_value`, `double step_value`)

The first argument in both cases must always be **this*, where *this* is the pointer to the sink object itself. The choice of which *initialize* function is called will determine the way the sink behaves:

1. When the first *initialize* function is called, the *SinkControl* object will add a task to the `TaskList` and reset its internal timer/counter to 0. Then every time the *collectData* function is called, it increments the timer/counter by *step_value* and returns 1 (TRUE) if the timer/counter (before being incremented) was between *start_value* and *stop_value*. It returns 0 (FALSE) otherwise. When the timer/counter reaches *stop_value* the task is automatically removed from the `TaskList`. If you want to remove the task from the `TaskList` before *stop_value* is reached, you can do so by calling the *stopControl* function. This is used in the BER sinks (see example `SimpleBERSink` at the end of this section) where *stopControl* is called when some condition is satisfied. Although in this case you might want to ignore *stop_value* completely, it still makes sense to define it as an upper limit of how long the simulation will run just in case the condition that needs to be satisfied before *stopControl* is called is never satisfied. If you are absolutely certain that the condition you are using will be satisfied, and do

not know what value to use for *stop_value*, use a very large value, e.g., 1.0e20. When calling this *initialize* function, *start_value* must be greater than or equal to 0, *stop_value* must be greater than or equal to *start_value*, and *step_value* must be greater than 0. If these conditions are not satisfied the simulation will abort.

2. When the second *initialize* function is called, the *SinkControl* object will not add a task to the TaskList. Therefore, the sink will not control how long the simulation will run. The timer/counter is still reset to 0. When *collectData* is called, it increments the timer/counter by *step_value* and returns 1 (TRUE) if the timer/counter (before being incremented) was greater than or equal to *start_value*. In this mode of operation, the sink can be used to collect all the data from a simulation controlled by some other sink or source. This may be useful for large, multirate designs, where you do not know the rates at the points where you want to collect the data. If the sink's *SinkControl* object is initialized using the first *initialize* function and *start_value*, *stop_value* are not selected appropriately, the design might end up simulating a lot more than it should. By setting only one sink to control the simulation and letting the others collect data as long as the simulation runs, you can guarantee that the data collected in all sinks will correspond time-wise to the data collected in the sink that controlled the simulation. Another case where this mode of operation is useful is when the input signal for a simulation is read from a file and the amount of data in the file is not known. By setting only the source to control the simulation, you can “force” the sinks to collect the right amount of data no matter how much data there is in the file. When calling this *initialize* function, *start_value* must be greater than or equal to 0 and *step_value* must be greater than 0. If these conditions are not satisfied, the simulation will abort.

Useful Notes/Hints

- A sink model need not operate in only one of the two ways described above. Parameters can be used to decide how the sink's *SinkControl* object is initialized. For example, this is the purpose of the *ControlSimulation* parameter of the *NumericSink* and *TimedSink* models. Also see the examples in the section “[Examples of Sink Models](#)” on page 14-51.
- When a sink is to be used with numeric data, it is recommended to use 1 as the *step_value* when calling the *initialize* function. When a sink is to be used with timed data, it is recommended to use the simulation time step (obtained by

calling `input.getTimeStep()`, where `input` is the name of the input port) as the *step_value* when calling the *initialize* function.

- It is recommended that you only write uni-rate sink models, that is, sinks that only read one sample from their inputs every time they are fired.
- A sink that needs to post-process the data it collects, that is, it just stores the data in some array during go and processes it in wrapup, must always be initialized using the first *initialize* function. Otherwise, you will not know how much memory needs to be allocated for the array that will store the collected data during go. Examples of sinks like that are the `SpectrumAnalyzer`, `EVM`, and refer to “[SDFMedianSink](#)” on page 14-52. 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 14-21 in this chapter. Refer to “[SDFSimpleBERSink](#)” on page 14-52 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)
- Cannot use the abstract class *SimData* as the type of an object, parameter type or return type (AIX)

What follows describes the use of the functions of the *SimData* class. While reading the following paragraphs keep in mind that data written to a dataset always has a dependent and independent variable associated with it.

- *newSimData*(Block **starP*). This is not a function of the *SimData* class but it must be called in order to initialize the pointer to the *SimData* class. This must be done before the pointer can be used. The argument of this function must always be *this*, where *this* is the pointer to the sink object itself. For example, if you have defined a pointer to an object of type *SimData* and its name is *dataP*, then the following piece of code should precede any use of *dataP*:

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

- *setIndepVar*(const char **name*, 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 *baseName.suffix*. This is useful when a sink writes multiple variables to the dataset.
- *setAutoPlotType*(const int &*type*) can be used to automatically plot data at the end of the simulation. If the value of 0 is passed, no automatic plotting occurs. If the value of 1 is passed, a rectangular plot is automatically plotted at the end of the simulation.
- *addAttribute*(const char **name*, int *value*) and *addAttribute*(const char **name*, double *value*) can be used to associate integer or double attributes with the data. For example, the TimedSink uses this function to associate a characterization frequency with its data. To retrieve the value of an attribute in the Data Display window, you have to use the function *get_attr*(sinkName, attributeName), e.g., char_freq = get_attr(T1, "fc").
- *sendData*(*x*, *y*) (six overloaded versions) is used to send data to the dataset.

Typically, a sink handles only one type of data. However, there is no such limitation. A sink can be written to handle any type of data by declaring its input to be of type *ANYTYPE*. To get the type of data that the sink has received in a particular simulation, the method *PortHole::resolvedType()* can be called. Then according to the value this method returns, you can call the *setDepVar()* and *sendData()* methods with the appropriate arguments to handle the specific data type the sink has received. Refer to the example “[SDFSimpleNumericSink](#)” on page 14-52, which can handle *INT*, *FLOAT*, and *COMPLEX* data.

Examples of Sink Models

This section gives links to five examples of sink models:

SDFSimpleNumericSink

TSDFSimpleTimedSink

SDFSimpleBERSink

SDFMeanVarianceSink

SDFMedianSink

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 *MeadianSink* is an example of a post-processing sink; it just collects the data in *go* and does all the processing as well as writes the results to the dataset in the *wrapup* method.

Chapter 15: 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 15-1](#). The fixed-point scalar type has two forms, UCB (University of California at Berkeley) fixed-point and HP fixed-point. The HP 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 15-19](#). 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 15-25](#).

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 15-40](#). There is also the timed signal type with two forms, Baseband and Complex Envelope, described in the section, “[Timed Particle Signal Type](#)” on [page 15-40](#).

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 14-24](#) in Chapter 14, 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:

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

Non-Member Functions and Operators

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

```
Complex operator + (const Complex& x, const Complex& y)  
Complex operator - (const Complex& x, const Complex& y)  
Complex operator * (const Complex& x, const Complex& y)  
Complex operator * (double x, const Complex& y)  
Complex operator * (const Complex& x, double y)  
Complex operator / (const Complex& x, const Complex& y)  
Complex operator / (const Complex& x, double y)  
Complex operator - (const Complex& x)
```

Return the negative of the complex number.

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

Return the complex conjugate of the number.

```
Complex sin(const Complex& x)  
Complex cos(const Complex& x)  
Complex exp(const Complex& x)  
Complex log(const Complex& x)  
Complex sqrt(const Complex& x)  
Complex pow(double base, const Complex& expon)  
Complex pow(const Complex& base, const Complex& expon)
```

Other general operators:

```
double abs(const Complex& x)
```

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

```
double arg(const Complex& x)
```

Return the value $\arctan(x.\text{imag}()/x.\text{real}())$.

```
double norm(const Complex& x)
```

Return the value $x.\text{real}() * x.\text{real}() + x.\text{imag}() * x.\text{imag}()$.

```
double real(const Complex& x)
```

Return the real part of the complex number.

```
double imag(const Complex& x)
```

Return the imaginary part of the complex number.

Comparison Operators:

```
int operator != (const Complex& x, const Complex& y)
```

```
int operator == (const Complex& x, const Complex& y)
```

The Fixed-Point Data Type

The fixed-point data type is implemented in 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:

$x.y$ or y/n

where

x = number of integer bits (including sign bit) to the left of the decimal point

y = number of fractional bits to the right of the decimal point

n = total number of bits (bitwidth)

Fixed-point operations include consideration of overflow type (wrapped, saturate, saturate-to-zero) and quantization type (round or truncate).

Agilent Ptolemy Fix class is an extension of the UCB Fix class. The distinction between the UCB and HP 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 15-1. Fix Data Type Properties

Attributes for Fix Data Type	UCB Fixed-Point Stars	HP Fixed-Point Stars
ArithType, default	2's complement	2's complement
ArithType, options.	2's complement	Synthesizable DSP—2's complement, unsigned Non-synthesizable DSP—2's complement
Precision, max bit width	32	256
Overflow handler, default	saturate	Synthesizable DSP—wrapped Non-synthesizable DSP—saturate
Overflow handler, options	wrapped, saturate, saturate-to-zero	Synthesizable DSP—wrapped, saturate Non-synthesizable DSP—wrapped, saturate, saturate-to-zero
RoundFix, default	truncate	truncate
RoundFix, options.	round, truncate	round, truncate
Generates Verilog or VHDL	No	Synthesizable DSP—Yes Non-synthesizable DSP—No

Constructing Fixed-Point Variables

Variables of type Fix are defined by specifying the word length and the position of the binary point. At the user-interface level, precision is specified either by setting a fixed-point parameter to a (*value, precision*) pair, or by setting a *precision* parameter. The former gives the value and precision of some fixed-point value, while the latter is typically used to specify the internal precision of star computations. In either case, the syntax of the precision is either $x.y$ or m/n , where x is the number of integer bits (including the sign bit), y and m are the number of fractional bits, and n is the total number of bits. Thus, the total number of bits in the fixed-point number (also called

its *length*) is $x + y$ or n . For example, a fixed-point number with precision 3.5 has a total length of 8 bits, with 3 bits to the left and 5 bits to the right of the binary point.

At the source code level, methods working on `Fix` objects either have the precision passed as an $x.y$ or m/n string, or as two C++ integers that specify the total number of bits and the number of integer bits including the sign bit (that is, n and x). For example, suppose you have a star with a precision parameter named *precision*. Consider the following code:

```
Fix x = Fix(((const char *) precision));
if (x.invalid())
    Error::abortRun(*this, "Invalid precision");
```

The *precision* parameter is cast to a string and passed as a constructor argument to the `Fix` class. The error check verifies that the precision was valid.

There is a maximum value for the total length of a `Fix` object which is 256 bits. Numbers in the `Fix` class represented using two's complement notation have the sign bit stored in the bits to the left of the binary point. There must always be at least one bit to the left of the binary point to store the sign for two's complement arithmetic type.

In addition to its value, each `Fix` object contains information about its precision and error codes indicating overflow, divide-by-zero, or bad format parameters. The error codes are set when errors occur in constructors or arithmetic operators. There are also fields to specify:

- Whether rounding or truncation takes place when other `Fix` values are assigned to it—truncation is the default;
- The response to an overflow or underflow on assignment—the default is saturation for UCB fixed-point, and wrapped for HP fixed-point (see [“Assignment and Overflow Handling” on page 15-9](#)).

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:

```
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_overflow(<keyword>)`. The keywords for overflow handling methods are: *saturate* (default), *zero_saturate*, *wrapped*, and *warning*. With *saturate*, the original value is replaced by the maximum (for overflow) or minimum (for underflow) value representable given the precision of the `Fix` object. *zero_saturate* sets the value to zero.

Explicitly Casting Inputs

In the above example, the first line of the `go` method assigned the input to the protected member `sum`, which has the side-effect of quantizing the input to the precision of `sum`. Alternatively, we could have written the `go` method as follows:

```
go {
    sum = Fix(input1%0) + Fix(input2%0);
}
```

```

    output%0 << sum;
}

```

The behavior here is significantly different: the inputs are added using their own native precision, and only the result is quantized to the precision of *sum*.

Some stars enable you to select between these two different behaviors with a parameter called *UseArrivingPrecision*. If set to *YES*, the input particles are not explicitly cast; they are used as they are; if set to *NO*, the input particles are cast to an internal precision, which is usually specified by another parameter.

Here is the (abbreviated) source of the *SDFGainFix* star, which demonstrates this point:

```

defstar {
  name {GainFix}
  domain {SDF}
  derivedFrom {SDFFix}
  desc {

```

This is an amplifier; the fixed-point output is the fixed-point input multiplied by the "gain" (default 1.0). The precision of "gain", the input, and the output can be specified in bits.

```

}
input {
  name {input}
  type {fix}
}
output {
  name {output}
  type {fix}
}
defstate {
  name {gain}
  type {fix}
  default {1.0}
  desc {Gain of the star.}
}
defstate {
  name {UseArrivingPrecision}
  type {int}
  default {"YES"}
  desc {

```

Flag indicating whether or no to use the arriving particles as they are: YES keeps the same precision, and NO casts them to the precision specified by the parameter "InputPrecision".)

```
}  
defstate {  
  name {InputPrecision}  
  type {precision}  
  default {2.14}  
  desc {
```

Precision of the input in bits. The input particles are only cast to this precision if the parameter "ArrivingPrecision" is set to NO.

```
}  
}  
defstate {  
  name {OutputPrecision}  
  type {precision}  
  default {2.14}  
  desc {
```

Precision of the output in bits. This is the precision that will hold the result of the arithmetic operation on the inputs. When the value of the product extends outside of the precision, the OverflowHandler will be called.

```
}  
protected {  
  Fix fixIn, out;  
}  
begin {  
  SDFFix::begin();  
  
  if (!int(UseArrivingPrecision)) {  
    fixIn = Fix(((const char *) InputPrecision));  
    if(fixIn.invalid())  
      Error::abortRun(*this, "Invalid InputPrecision");  
  }  
  
  out = Fix(((const char *) OutputPrecision));  
  if (out.invalid())  
    Error::abortRun(*this, "Invalid OutputPrecision");  
  out.set_ovflow(((const char*) OverflowHandler.enumString ((int)  
OverflowHandler)));
```

```

    if(out.invalid())
        Error::abortRun(*this, "Invalid OverflowHandler");
}
go {
    // all computations should be performed with out since
    // that is the Fix variable with the desired overflow
    // handler
    out = Fix(gain);
    if (int(UseArrivingPrecision)) {
        out *= Fix(input%0);
    }
    else {
        fixIn = Fix(input%0);
        out *= fixIn;
    }
    checkOverflow(out);
    output%0 << out;
}

// a wrap-up method is inherited from 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.

Fix(const char* precisionString, uint16* bits)

Create a Fix with the specified precision and set the bits precisely to the ones in the given *bits*. The first word pointed to by *bits* contains the most significant 16 bits of the representation. Only as many words as are necessary to fetch the bits will be referenced from the *bits* argument. For example: *Fix("2.14",bits)* will only reference *bits[0]*.

Fix(const Fix& arg)

Copy constructor. Produces an exact duplicate of *arg*.

Fix(int length, int intbits, const Fix& arg)

Read the value from the Fix argument and set to a new precision. If the precision parameters are not valid, then an error bit is internally set so that the *invalid*

method will return TRUE when called on the object. If the value from the source will not fit, an error bit is set so that the `ovf_occurred` method will return TRUE.

Functions to Set Or Display Information about the Fix Number

`int len() const`

Returns total word length of the Fix number.

`int intb() const`

Returns number of bits to the left of the binary point.

`int precision() const`

Returns number of bits to the right of the binary point.

`int overflow() const`

Returns the code of the overflow response type for the Fix number. The possible codes are:

0 = `ovf_saturate`

1 = `ovf_zero_saturate`

2 = `ovf_wrapped`

3 = `ovf_warning`

4 = `ovf_n_types`

`int roundMode() const`

Returns the rounding mode: *1* for rounding, *0* for truncation.

`int signBit() const`

Returns TRUE if the value of the Fix number is negative, FALSE if it is positive or zero.

`int is_zero()`

Returns TRUE if the value of Fix is zero.

`double max()`

Returns the maximum value representable using the current precision.

`double min()`

Returns the minimum value representable using the current precision.

double value()

The value of the Fix number as a double.

void setToZero()

Sets the value of the Fix number to zero.

void set_overflow(int value)

Sets the overflow type.

void set_rounding(int value)

Sets the rounding type: TRUE for rounding, FALSE for truncation.

void initialize()

Discards the current precision format and set the Fix number to zero.

There are a few functions for backward compatibility:

void set_ovflow(const char*)

Sets the overflow using a name.

void Set_MASK(int value)

Sets the rounding type. Same functionality as *set_rounding()*.

Comparison function:

int compare (const Fix& a, const Fix& b)

Compares two Fix numbers. Return -1 if $a < b$, 0 if $a = b$, 1 if $a > b$.

The following functions are for use with the error condition fields:

int ovf_occurred()

Returns TRUE if an overflow has occurred as the result of some operation like addition or assignment.

int invalid()

Returns TRUE if the current value of the Fix number is invalid due to it having an improper precision format, or if some operation caused a divide by zero.

int dbz()

Returns TRUE if a divide by zero error occurred.

```
void clear_errors()
```

Resets all error bit fields to zero.

Operators

```
Fix& operator = (const Fix& arg)
```

Assignment operator. If **this* does not have its precision format set (that is, it is uninitialized), the source Fix is copied. Otherwise, the source Fix value is converted to the existing precision. Either truncation or rounding takes place, based on the value of the rounding bit of the current object. Overflow results either in saturation, “zero saturation” (replacing the result with zero), or a warning error message, depending on the overflow field of the object. In these cases, `ovf_occurred` will return TRUE on the result.

```
Fix& operator = (double arg)
```

Assignment operator. The double value is first converted to a default precision Fix number and then assigned to **this*.

The function of these arithmetic operators should be self-explanatory:

```
Fix& operator += (const Fix&)
```

```
Fix& operator -= (const Fix&)
```

```
Fix& operator *= (const Fix&)
```

```
Fix& operator *= (int)
```

```
Fix& operator /= (const Fix&)
```

```
Fix operator + (const Fix&, const Fix&)
```

```
Fix operator - (const Fix&, const Fix&)
```

```
Fix operator * (const Fix&, const Fix&)
```

```
Fix operator * (const Fix&, int)
```

```
Fix operator * (int, const Fix&)
```

```
Fix operator / (const Fix&, const Fix&)
```

```
Fix operator - (const Fix&) // unary minus
```

```
int operator == (const Fix& a, const Fix& b)
```

```
int operator != (const Fix& a, const Fix& b)
```

`int operator >= (const Fix& a, const Fix& b)`

`int operator <= (const Fix& a, const Fix& b)`

`int operator > (const Fix& a, const Fix& b)`

`int operator < (const Fix& a, const Fix& b)`

Notes:

- These operators are designed so that overflow does not, as a rule, occur (the return value has a wider format than that of its arguments). The exception is when the result cannot be represented in a *Fix* with all 64 bits before the binary point.
- The output of any operation will have error codes that are the logical OR of those of the arguments to the operation, plus any additional errors that occurred during the operation (like divide by zero).
- The division operation is currently a cheat: it converts to double and computes the result, converting back to *Fix*.
- The relational operators `==`, `!=`, `>=`, `<=`, `>`, `<` are all written in terms of a function:

`int compare(const Fix& a, const Fix& b)`

Returns -1 if $a < b$, 0 if $a = b$, and 1 if $a > b$. The comparison is exact (every bit is checked) if the two values have the same precision format. If the precisions are different, the arguments are converted to doubles and compared. Since *double* values only have an accuracy of about 53 bits on most machines, this may cause false equality reports for *Fix* values with many bits.

Conversions

`operator int() const`

Returns the value of the *Fix* number as an integer, truncating towards zero.

`operator float() const`

`operator double() const`

Converts to a float or a double, creating an exact result when possible.

`void complement()`

Replaces the current value by its complement.

Fix Overflow, Rounding, and Errors

The `Fix` class defines the following enumerated values for overflow handling:

```
Fix::ovf_saturate
```

```
Fix::ovf_zero_saturate
```

```
Fix::ovf_wrapped
```

```
Fix::ovf_warning
```

These can be used as arguments to the `set_overflow` method, as in the following example:

```
out.set_overflow(Fix::ovf_saturate)
```

The member function

```
int overflow() const
```

returns the overflow type. This returned result can be compared against the above enumerated values. Overflow types may also be specified as strings, using the following method:

```
void set_ovflow(const char* overflow_type);
```

the `overflow_type` argument may be one of “saturate”, “zero_saturate”, “wrapped”, or “warning”.

The rounding behavior of a `Fix` value may be set by calling:

```
void set_rounding(int value);
```

If the argument is false, or has the value `Fix::mask_truncate`, truncation will occur. If the argument is nonzero (for example, if it has the value `Fix::mask_truncate_round`), rounding will occur. The older name `Set_MASK` is a synonym for `set_rounding`.

The following functions access the error bits of a `Fix` result:

```
int ovf_occurred() const
```

```
int invalid() const
```

```
int dbz() const
```

The first function returns `TRUE` if there have been any overflows in computing the value. The second returns `TRUE` if the value is invalid, because of invalid precision parameters or a divide by zero. The third returns `TRUE` only for divide by zero.

Defining New Data Types

The 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.
```

```

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;

```



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

#endif /*VECTOR_H_INCLUDED*/
```

Example: Using the Vector Message Class in a Custom Model

To build a new model using the *Vector* message type example:

1. Create new directory for model development as outlined in [Chapter 13, Building Signal Processing Models](#).
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 13, 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:

<type>: no print method

where *type* is whatever is returned by `type()`.

By redefining these methods, you can make it legal to connect a star that generates messages to a star that expects integer, floating, or complex particles, or you can connect to a *Printer* star.

Use of the Envelope Class

The Envelope class references objects of class Message or its derived classes. Once a Message object is placed into an Envelope object, the Envelope takes over responsibility for managing its memory, that is, maintaining reference counts and deleting the message when it is no longer needed.

The constructor (which takes as its argument a reference to a Message), copy constructor, assignment operator, and destructor of *Envelope* manipulate the reference counts of the reference's *Message* object. Assignment simply copies a pointer and increments the reference count. When an Envelope destructor is called, the reference count of the Message object is decremented; if it becomes zero, the Message object is deleted. Because of this deletion, a Message must never be put inside an Envelope unless it was created with the *new* operator. Once a Message object is put into an Envelope it must never be explicitly deleted; it will “live” as long as there is at least one Envelope that contains it.

It is possible for an Envelope to be *empty*. If it is, the empty method will return *TRUE*, and the data field will be *NULL*.

The *type* method of Envelope returns the datatype of the contained Message object. To access the data, the following two methods are provided:

- The *myData* function returns a pointer to the contained Message-derived object.

Note *The data pointed to by this pointer must not be modified, since other Envelope objects in the program may also contain it. If you convert its type, always make sure that the converted type is a pointer to const (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 15-19](#).

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 15-27](#).

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. HP 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 numRows, int numCol)

Example: *FloatMatrix A(3,2)*.

Create a matrix with dimensions *numRow* by *numCol*. Memory is allocated for data storage but the entries are uninitialized.

XXXMatrix(int numRows, 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 numRows, 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 numRows, 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 $startRow + numRows$ 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 numRows, int numCol, int length, int intBits)`

Example: *FixMatrix A(2,2,14,4)*.

Create a *FixMatrix* with the given dimensions such that each entry is a fixed-point number with precision as given by the *length* and *intBits* inputs.

`FixMatrix(int numRows, int numCol, int length, int intBits, PortHole& myPortHole)`

Example: *FixMatrix A(2,2,14,4)*.

Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the *length* and *intBits* inputs and initialized with the values that are read from the particles contained in the porthole *myPortHole*.

```
FixMatrix(int numRows, int numCol, int length, int intBits, FixArrayState&
dataArray)
```

Example: *FixMatrix A(2,2,14,4)*.

Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the *length* and *intBits* inputs and initialized with the values in the given `FixArrayState`.

There are also special copy constructors for the `FixMatrix` class that enable the programmer to specify the precision of the entries of the `FixMatrix` as they are copied from the sources. These copy constructors are usually used for easy conversion between the other matrix types. The last argument specifies the type of masking function (truncate, rounding, etc.) to be used when doing the conversion.

```
FixMatrix(const XXXMatrix& src, int length, int intBits,
int round)
```

Example: *FixMatrix A(CxMatrix,4,14,TRUE)*.

Create a `FixMatrix` with the given dimensions such that each entry is a fixed-point number with precision as given by the *length* and *intBits* arguments. Each entry of the new matrix is copied from the corresponding entry of the *src* matrix and converted as specified by the *round* argument.

Comparison Operators

```
int operator == (const XXXMatrix& src)
```

Example: *if(A == B) then ...*

Return `TRUE` if the two matrices have the same dimensions and every entry in *A* is equal to the corresponding entry in *B*. Return `FALSE` otherwise.

```
int operator != (const XXXMatrix& src)
```

Example: *if(A != B) then ...*

Return `TRUE` if the two matrices have different dimensions or if any entry of *A* differs from the corresponding entry in *B*. Return `FALSE` otherwise.

Conversion Operators

Each matrix class has a conversion operator so that the programmer can explicitly cast one type of matrix to another (this casting is done by copying). It would have been possible to make conversions occur automatically when needed, but because these conversions can be quite expensive for large matrices, and because unexpected results might occur if the user did not intend for a conversion to occur, we chose to require that these conversions be used explicitly.

operator XXXMatrix () const

Example: $FloatMatrix = A * (FloatMatrix)B$.

Convert a matrix of one type into another. These conversions allow the various arithmetic operators, such as $*$ and $+$, to be used on matrices of different types. For example, if A in the example above is a (3,3) FloatMatrix and B is a (3,2) IntMatrix, then C is a FloatMatrix with dimensions (3,2).

Destructive Replacement Operators

These operators are member functions that modify the current value of the object. In the following examples, A is usually the lvalue (**this*). All operators return **this*:

XXXMatrix& operator = (const XXXMatrix& src)

Example: $A = B$.

This is the assignment operator: make A into a matrix that is a copy of B . If A already has allocated data storage, then the size of this data storage is compared to the size of B . If they are equal, then the dimensions of A are simply set to those of B and the entries copied. If they are not equal, the data storage is freed and reallocated before copying.

XXXMatrix& operator = (xxx value)

Example: $A = value$.

Assign each entry of A to have the given *value*. Memory management is handled as in the previous operator. *Warning: this operator is targeted for deletion. Do not use it.*

XXXMatrix& operator += (const XXXMatrix& src)

Example: $A += B$.

Perform the operation $A.entry(i) += B.entry(i)$ for each entry in A . A and B must have the same dimensions.

`XXXMatrix& operator += (xxx value)`

Example: $A += value$.

Add the scalar *value* to each entry in the matrix.

`XXXMatrix& operator -= (const XXXMatrix& src)`

Example: $A -= B$.

Perform the operation $A.entry(i) -= B.entry(i)$ 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.

`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: $B = \sim A$.

Return a new matrix that is the transpose of the source.

XXXMatrix operator ! ()

Example: $B = !A$.

Return a new matrix that is the inverse of the source.

XXXMatrix operator ^ (int exponent)

Example: $B = A^2$.

Return a new matrix that is the source matrix to the given *exponent* power. The exponent can be negative, in which case it is first treated as a positive number and the final result is then inverted. So $A^2 == A*A$ and $A^{(-3)} == !(A*A*A)$.

XXXMatrix transpose()

Example: $B = A.transpose()$.

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.

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()

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)

Return the number of rows in the matrix.

int numCols()

Return the number of columns in the matrix.

Message* clone()

Example: $IntMatrix *B = A.clone()$.

Return a copy of **this*.

StringList print()

Example: $A.print()$.

Return a formatted StringList that can be printed to display the contents of the matrix in a reasonable format.

`XXXMatrix& multiply (const XXXMatrix& left, const XXXMatrix& right, XXXMatrix& result)`

Example: *multiply(A,B,C)*.

This is a faster 3-operand form of matrix multiply such that the result matrix is passed as an argument in order to avoid the extra copy step involved when writing $C = A * B$.

`const char* dataType()`

Example: *A.dataType()*.

Return a string that specifies the name of the matrix type. Available strings are ComplexMatrix, FixMatrix, FloatMatrix, and IntMatrix.

`int isA(const char* type)`

Example: *if(A.isA("FixMatrix")) then ...*

Return TRUE if the argument string matches the type string of the matrix.

Writing Stars and Programs Using the PtMatrix Class

This section describes how to use the matrix data classes when writing stars. Some examples are given here. For more examples, refer to the stars in `$PTOLEMY/src/domains/sdf/matrix/stars/*.pl` and `$PTOLEMY/src/domains/sdf/image/stars/*.pl`.

Memory Management

The most important thing to understand about the use of matrix data classes in the 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

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

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

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

is sufficient.

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

```
input {
    name {inputPortHole}
    type {FLOAT_MATRIX}
}
```

The syntax is the same for output portholes. The `type` field can be `COMPLEX_MATRIX`, `FLOAT_MATRIX`, `FIX_MATRIX`, or `INT_MATRIX`.

The icons created by 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:

```
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 -24.) 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 -24.

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 on page -20).

Matrix Outputs

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

```
FloatMatrix& outMatrix =*(new FloatMatrix(someRow,someCol));
// ... do some operations on the outMatrix
outputPortHole%0 << outMatrix;
```

The last line is similar to that for outputting a scalar value. This is because we have overloaded the *<< operator* for *MatrixEnvParticles* to support *PtMatrix* class inputs. The standard use of the *MessageParticle* class requires you to put your message into an envelope first and then use *<<* on the envelope (see “Use of the Envelope Class” on page -24), but we have specialized this so that the extra operation of creating an envelope first is not explicit.

The following is an example of a complete star definition that inputs and outputs matrices:

```
defstar {
  name {Add_M}
  domain {SDF}
```



```

desc {Output is the sum of all the floating-point input matrices.}
location {Numeric Matrix}
inmulti {
  name {input}
    type {FLOAT_MATRIX}
}
output {
  name {output}
  type {FLOAT_MATRIX}
}
ccinclude {"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:

```
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), flavor, Fc, and t}

where $i(t)$ and $q(t)$ are the timed signal in phase and quadrature components, $flavor$ represents a modulated signal, F_c is the carrier (or characterization) frequency, and t is the time.

There are two equivalent representations (*flavors*) of a timed signal:

complex envelope (ComplexEnv) $v(t)$

real baseband (BaseBand) $V(t)$

RF signals that are represented in the ComplexEnv flavor $v(t)$ together with F_c can be converted to the real BaseBand flavor $V(t)$ as:

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

Constructors

TimedParticle(const Complex& cx, const double& t, const double& fc);

Creates a Timed particle with a cx data, at time t , centered around F_c . Flavor is set to Complex Envelope.

TimedParticle(const double& x, const double& t);

Creates a Timed particle with data x , at time t . The F_c is set to zero and the flavor to Baseband.

TimedParticle(const int& x, const double& t);

Creates a Timed particle with data x , at time t . The 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 ptlang file. This example also demonstrates how to use the setup method to set the output carrier frequency and how to use begin method to set the default time step available at the output port.

```
output {
  Name { out1 }
  Type { Timed }
  Desc { Timed source output signal }
}
defstate {
  name { TStep }
  type { float }
```

```

default { 0.001 }
desc { Simulation time step; use a value of 0 for time step synchronization with
other network timed signals }
units { TIME_UNIT }
attributes { A_SETTABLE | A_NONCONSTANT }

}
defstate {
name { FCarrier }
type { float }
default { 1000000 }
desc { Output signal carrier frequency } units { FREQUENCY_UNIT }
attributes { A_SETTABLE | A_NONCONSTANT }
}
setup {
out1.setTimeStep( double(TStep));
out1.setCarrierFrequency( double(FCarrier));
}
begin {
if ( double(TStep) == 0.) {
double timestep;
timestep = out1.getTimeStep();
TStep = timestep;
}
out1.setTimeStep( double(TStep));
}

```

Carrier Frequency Resolution

For Timed data types, many times the carrier frequency must be propagated from the inputs to the outputs.

Each Timed user-compiled component can define a custom way of propagating the carrier frequency from the inputs to the outputs. By default, each output is marked with the maximum F_c available at all of the inputs. To override this method, use the following ptlang template. The following template can be modified to meet the specific component requirement for output carrier frequency:

```

method {
name {propagateFc }

```

```

access { protected }
arglist { "(double*)" }
type { void }
code {
    // Create an iterator for the ports of this star
    BlockPortIter nextPort(*this);
    TSDFPortHole* port;

    // Find the maximum fc over all of the inputs
    double maxFc = 0;
    while ((port = (TSDFPortHole*)nextPort++) != NULL) {
        // Ignore unconnected ports and output ports
        if (!(port->far() || port->isItOutput())) continue;
        double fc = port->getCarrierFrequency();
        if (fc > maxFc) maxFc = fc;
    }

    // Reset the iterator
    nextPort.reset();

    // Now set all Timed output ports to the maximum Fc
    // found over all of the inputs
    while ((port = (TSDFPortHole*)nextPort++) != NULL) {
        // Ignore unconnected ports and input ports
        if (!(port->far() ||
            port->isItInput() ||
            (port->resolvedType() != TIMED)))
            continue;
        port->setCarrierFrequency(maxFc);
    }
}
}
}

```


Chapter 16: 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 straight forward. 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 13, 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 13, 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 8.0 whereas UCB Ptolemy uses Tcl 7.6 / Tk 7.4 with iTcl extensions. There are many advantages in using Tcl/Tk 8.0 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 8.0 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 15-19 in Chapter 15, 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 15-19 in Chapter 15, 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 [“Defining New Data Types” on page 15-19 in Chapter 15, 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 15-20 in Chapter 15, *Data Types for Model Builders*, for more information. This should be a relatively straight forward procedure.

- Optional: Search and replace the MESSAGE port data type in your star library with the data type appropriate for your message.
- Optional: Create type conversion stars, using the type conversion ptlang keyword in place of the defstar directive. This will enable 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`.

Chapter 17: Glossary

Terminology

Throughout most of the Agilent Ptolemy documentation, we use the Advanced Design System terminology, which is standard EDA terminology. However, UC Berkeley Ptolemy has its own terminology and for users familiar with this terminology, or those who are writing their own models, the following table compares the terms. In the chapters on building signal processing models and in [Chapter 9, Theory of Operation](#), the UC Berkeley Ptolemy terminology used; elsewhere the Advanced Design System terminology is used. The remainder of the Glossary defines the various terms.

Table 17-1. Terminology Comparison

Agilent Ptolemy	UC Berkeley Ptolemy
Component	Star
Network (or circuit)	Galaxy
Top-level System	Universe
Controller	Target
Wire	Arc
Data (or signals)	Particles (or tokens)

\$HPEESOF_DIR

In UNIX installations, the environment variable specifying the directory in which the Advanced Design System software is installed. In Windows installations, the syntax, when needed, is `%HPEESOF_DIR%`.

actor

An atomic (indivisible) function in a dataflow model of computation. An actor is called a component in Agilent Ptolemy and a star in UCB Ptolemy.

arc

A wire that connects the output of one star or component with the input of another.

base class

A C++ object used to define common interfaces and common code for a set of derived classes. An object may be a base class and a derived class simultaneously.

behavioral modeling

System modeling consisting of functional specification plus modeling of the timing of an implementation (compare to functional modeling).

Block

The base class defined in the kernel for stars, galaxies, universes, and targets.

block

A star or a galaxy.

compile-time scheduling

A scheduling policy in which the order of block execution is pre-computed when the execution is started. The execution of the blocks thus involves only sequencing through this pre-computed order one or more times (compare to run-time scheduling).

derived class

A C++ object derived from some base class. It inherits all of the members and methods of the base class.

dataflow

A model of computation in which actors process streams of tokens. Each actor has one or more firing rules. Actors that are enabled by a firing rule may fire in any order.

domain

A specific implementation of a computation model.

Domain

The base class in the Agilent Ptolemy kernel from which all domains are derived.

drag

The action of holding a mouse button while moving the mouse.

FFT

The Fast Fourier Transform (FFT) is an efficient way to implement the discrete Fourier transform in digital hardware.

firing

A unit invocation of an actor in a dataflow model of computation.

firing rule

A rule that specifies how many tokens are required on each input of a dataflow actor for that actor to be enabled for firing.

fork star

A star that reads one input particle and replicates it on any number of outputs.

functional modeling

System modeling that specifies input/output behavior without specifying timing (compare to behavioral modeling).

galaxy

A block that contains a network of other blocks.

Gantt chart

A graphical display of a parallel schedule of tasks. In Agilent Ptolemy, the tasks are the firings of stars and galaxies.

homogeneous synchronous dataflow

A particular case of the synchronous dataflow model of computation, where actors produce and consume exactly one token on each input and output.

hpeesoflang

(1) A schema language used to define stars in Agilent Ptolemy. (2) The program that translates stars written in the hpeesoflang language to C++. In UCB Ptolemy, the equivalent language is called ptlang.

iteration

A set of executions of blocks that constitutes one pass through the pre-computed order of a compile-time schedule.

kernel

The set of classes defined in the Agilent Ptolemy kernel.

layer

In the Schematic, a color with a given precedence. Colors with higher precedence will obscure colors with lower precedence.

member

A C++ object that forms a portion of another object.

method

A function defined to be part of an object in C++.

model of computation

A set of semantic rules defining the behavior of a network of blocks.

net

A graphical connection between ports in the schematic.

object

A data type in C++ consisting of members and methods. These members and methods may be private, protected, or public. If they are private, they can only be accessed by methods defined in the object. If they are protected, they can also be accessed by methods in derived classes. If they are public, they can be accessed by any C++ code.

palette

A schematic area that contains a library of block icons.

parameter

The initial value of a state.

particle

Data (for example, a floating-point value) communicated between blocks.

port

A star or galaxy input or output.

PortHole

The base class in the Agilent Ptolemy kernel for all ports.

Ptolemy

A design environment that supports simultaneous mixtures of different computation models. Ptolemy, named after the second-century Greek astronomer, mathematician, and geographer, was developed at the University of California at Berkeley.

real time

The actual time (compare to simulated time).

RTL

Register-transfer level description of digital systems. This kind of description is used by DSP Synthesis.

run-time scheduling

A scheduling policy in which the order of block execution is determined “on-the-fly,” as they are executed (compare to compile-time scheduling).

Scheduler

An object associated with a domain that determines the order of block execution within the domain. Domains may have multiple schedulers.

schematic

A block diagram.

SDF

A simulation domain using the synchronous dataflow model of computation.

simulated time

In a simulation domain, the real number representing time in the simulated system (compare to real time).

simulation

The execution of a system specification (an Agilent Ptolemy block diagram) from within the Agilent Ptolemy process (that is, execution without generating code and spawning a new process to execute that code).

simulation domain

A domain that supports simulation, but not code generation.

star

A component in Agilent Ptolemy. An atomic (indivisible) unit of computation in an Agilent Ptolemy application. Every Agilent Ptolemy simulation ultimately consists of executing the methods of the stars used to define the simulation.

Star

The base class in the Agilent Ptolemy kernel for all stars.

state

A member of a block that stores data values from one invocation of the block to the next.

State

The base class in the Agilent Ptolemy kernel for all states.

stop time

Within a timed domain, the time at which a simulation halts.

symbol

A graphical object that represents a single block.

synchronous dataflow

A dataflow model of computation where the firing rules are particularly simple. Every input of every actor requires a fixed, pre-specified number of tokens for the actor to fire. Moreover, when the actor fires, a fixed, pre-specified number of tokens is produced on each output. This model of computation is particularly well-suited to compile-time scheduling.

target

An object that manages the execution of a simulation or code generation process. In Agilent Ptolemy this is called a controller. For example, in code generation, the target would be responsible for compiling the generated code and spawning the process to execute that code.

Target

The base class in the kernel for all targets.

Tcl

Tool command language—a textual, interpreted language developed by John Ousterhout at UC Berkeley. Tcl is embedded in Agilent Ptolemy.

timestamp

A real number associated with a particle in timed domains that indicates the point in simulated time at which the particle is valid.

timed domain

A domain that models the evolution of a system in time.

Tk

A Windows and X-Windows toolkit for Tcl. The interactive sliders, buttons, and plotting capabilities of Agilent Ptolemy are implemented in Tcl/Tk.

token

A unit of data in a dataflow model of computation. Tokens are implemented as particles in Agilent Ptolemy.

universe

An entire Agilent Ptolemy design.

VHDL

The VHSIC hardware description language, a standardized language for specifying hardware designs at multiple levels of abstraction.

wormhole

A star in a particular domain that internally contains a galaxy in another domain.

Index

A

- A_CONSTANT attribute, 14-12
- A_NONCONSTANT attribute, 14-12, 14-29
- A_NONSETTABLE attribute, 14-12
- A_SETTABLE attribute, 14-12, 14-29
- accessMessage method
 - MessageParticle class, 15-25
- Add (SDF block), 14-27
- AddFix (SDF block), 15-7
- Advanced Design System
 - approach to cosimulation, 11-8
- aggressive reclamation, 15-25
- anytype portholes, 14-15
- array parameters, 4-17
- ArrayState class, 14-30
- ArrivingPrecision parameter, 15-10
- arrowheads in schematic, 5-2
- attributes
 - for HP Ptolemy model development, 14-8, 14-10, 14-12, 14-29

B

- bad format parameters
 - Fix class, 15-6
- Baseband Equivalent Channel, 12-13
- binary point, 15-6

C

- carrier frequency resolution, 9-9
- ccinclude hpeesoflang directive, 14-19
- Circuit Envelope simulator
 - selecting waveform for cosimulation, 11-10
- circuit subnetworks, consequences of
 - clustering, 11-5
- clearAttributes method, 14-32
- clone method
 - Message class, 15-23, 15-25
- clustering
 - consequences of, 11-5
 - defined for cosimulation, 11-4
 - simulating resistors, 11-5
- code hpeesoflang directive, 14-19
- compile-time scheduling, 14-17
- Complex class, 14-28, 14-30, 15-2-15-4
 - operator, 15-3

- != operator, 15-4
- * operator, 15-3
- *= operator, 15-2
- + operator, 15-3
- += operator, 15-2
- / operator, 15-3
- /= operator, 15-2
- = operator, 15-2
- = operator, 15-2
- == operator, 15-4
- abs() function, 15-3
- arg() function, 15-4
- basic operators, 15-2
- conj() function, 15-3
- constructors, 15-2
- cos() function, 15-3
- exp() function, 15-3
- imag() function, 15-3, 15-4
- log() function, 15-3
- norm() function, 15-4
- pow() function, 15-3
- real() function, 15-3, 15-4
- sin() function, 15-3
- sqrt() function, 15-3
- Complex data type, 15-2-15-4
- complex state, 14-11
- complex type
 - portholes, 14-15
 - states, 14-10
- COMPLEX_MATRIX_ENV, 15-37
- complex_matrix_env type
 - portholes, 14-15
- complexarray type
 - states, 14-10
- ComplexArrayState class, 14-29
- ComplexMatrix, *See* Matrix class
- ComplexParticle class, 14-28
- ComplexState class, 14-29
- complex-valued parameters, 4-9
- conscalls hpeesoflang directive, 14-16
- constructor hpeesoflang directive, 14-16
- constructors, 14-16
- copy constructor
 - Message class, 15-23
- cosimulation

- clustering, consequences of, 11-5
- clustering, defined, 11-4
- example of, 11-13
- example results, 11-17
- nested simulation approach, 11-8
- resolving deadlocks, 11-6
- selecting circuit envelope waveform, 11-10
- time step sizes, 11-9
- troubleshooting common problems, 11-12
- using EnvOutSelector, 11-11
- using EnvOutShort, 11-11
- using time converters, 11-10
- using time steps, 11-9

D

- data types, 14-15
 - conversion of, 5-5
 - conversion, automatic or manual, 5-8
 - numeric matrix, 5-4
 - numeric scalar, 5-3
 - representation, 5-1
 - timed data, 5-4
 - user-defined, 15-19
- dataflow terminology, 9-2
- deadlocks, 9-7
 - resolving for cosimulation, 11-6
- default parameter values, 14-11
- default value for states, 14-11
- delay
 - for matrix arcs, 15-38
 - in dataflow, 15-38
- descriptor, 14-12
- destructor hpeesoflang directive, 14-17
- divide by zero
 - Fix class, 15-6
- dummy message, 15-25

E

- empty method
 - Envelope class, 15-24
- enumerated states, 14-10, 14-11
- Envelope class, 15-19, 15-24
- EnvOutSelector, using for cosimulation, 11-11
- EnvOutShort, using for cosimulation, 11-11
- examples of sink models, 14-51

F

- filename
 - parameters, 4-17
- filename states, 14-10, 14-11
- Fix class, 15-4-15-18
 - operator, 15-16
 - * operator, 15-16
 - *= operator, 15-16
 - + operator, 15-16
 - += operator, 15-16
 - / operator, 15-16
 - /= operator, 15-16
 - = operator, 15-16
 - = operator, 15-16
 - clear_errors(), 15-16
 - compare(), 15-15
 - complement(), 15-17
 - constructors, 15-12
 - conversion operators, 15-17
 - dbz(), 15-15
 - intb(), 15-14
 - invalid(), 15-15
 - is_zero(), 15-14
 - len(), 15-14
 - max(), 15-14
 - maximum length, 15-6
 - min(), 15-14
 - overflow(), 15-14
 - ovf_occurred, 15-15
 - precision(), 15-14
 - roundMode(), 15-14
 - set_overflow, 15-15
 - set_rounding, 15-15
 - setToZero(), 15-15
 - signBit(), 15-14
 - uninitialized, 15-9
 - value(), 15-15
- fix type
 - portholes, 14-15
 - states, 14-10
- FIX_MATRIX_ENV, 15-37
- fix_matrix_env type
 - portholes, 14-15
- fixed-point, 15-4
 - array parameters, 15-6
 - inputs and outputs, 15-7
 - parameters, 15-6
 - precision, 14-11

- setting precision, 14-11
 - states, 15-6
- fixed-point data type, 15-4
- FixMatrix, *See* Matrix class
- float type
 - portholes, 14-15
 - states, 14-10
- FLOAT_MATRIX_ENV, 15-37
- float_matrix_env type
 - portholes, 14-15
- floatarray type
 - states, 14-10
- FloatArrayState class, 14-29, 14-30
- FloatMatrix, *See* Matrix class
- FloatParticle class, 14-28
- FloatState class, 14-29
- Fork (SDF block), 14-27

G

- Gain (SDF block), 14-37
- getMessage method
 - MessageParticle class, 15-25
- go hpeesoflang directive, 14-18

H

- header hpeesoflang directive, 14-19
- heterogeneous message interface, 15-19
- hinclude hpeesoflang directive, 14-19
- homogeneous synchronous dataflow, 17-3
- HP Ptolemy
 - arrowheads in schematic, 5-2
 - data types, conversion, 5-5
 - data types, representation, 5-1
 - Interactive Controls and Displays,
 - compared to sinks, 12-1
 - overview versus UC Berkeley Ptolemy, 1-1
 - porting UC Berkeley models into, 16-1
 - references, 9-10
 - terminology versus UCB Ptolemy, 1-2
 - theory of operation, 9-1
- hpeesoflang *See* model development, 14-17

I

- initialized Fix objects, 15-9
- initializing states from files, 14-30
- inmulti hpeesoflang directive, 14-15, 14-26
- input hpeesoflang directive, 14-15, 14-24

- InSDFPort class, 14-26
- int type
 - portholes, 14-15
 - states, 14-10
- INT_MATRIX_ENV, 15-37
- int_matrix_env type
 - portholes, 14-15
- intarray type
 - states, 14-10
- IntArrayState class, 14-29
- Interactive Control and Display components
 - table, 12-2
- IntMatrix, *See* Matrix class
- IntParticle class, 14-28
- IntState class, 14-29
- Invoke Tcl Script, 12-14

L

- LMS Adaptive Filter, 12-9

M

- MATLAB
 - cosimulation, 10-1
 - examples, 10-4
 - setting up, 10-1
 - simulating with, 10-2
 - writing function for, 10-2
- Matrix class, 15-25-15-39
 - operator, 15-34
 - operator, unary negation operator, 15-33
 - ! operator, inverse operator, 15-33
 - != operator, 15-30
 - * operator, 15-34
 - *= operator, 15-32
 - + operator, 15-34
 - += operator, 15-31
 - /= operator, 15-32
 - = operator, 15-32
 - = operator, assignment operator, 15-31
 - == operator, 15-30
 - ^ operator, 15-33
 - ~ operator, transpose operator, 15-33
 - clone() function, 15-35
 - ComplexMatrix, 15-27
 - conjugate() function for ComplexMatrix, 15-33
 - constructors, 15-28
 - conversion operators, 15-31

- dataType() function, 15-36
- entry() function, 15-28
- FixMatrix, 15-27
- FixMatrix, special constructors, 15-30
- FloatMatrix, 15-27
- hermitian() function for ComplexMatrix, 15-33
- including Matrix.h into a Star, 15-37
- identity() function, 15-32
- IntMatrix, 15-27
- inverse() function, 15-33
- isA() function, 15-36
- multiply() function, 15-36
- outputting to a PortHole, 15-38
- print() function, 15-35
- star input and output, 15-37
- transpose() function, 15-33
- writing Stars that use the Matrix class, 15-36

Matrix.h include file, 15-37

memory leaks, preventing, 14-39

Message class, 15-19

message data type, 14-15

message type

- portholes, 14-15

MessageParticle class, 14-28, 15-19, 15-25

method hpeesoflang directive, 14-19

model development

- signal processing
 - command-line method, 13-1
 - creating a simple model library, 13-3
 - debugging, 13-10
 - platform-specific issues, 13-12
 - prerequisites to model development, 13-2
 - src directory and make-defs in detail, 13-8
 - write a model, 13-10

N

- nominal optimization
 - See optimization, 8-1

- numberPorts method, 14-28
- numtokens hpeesoflang directive, 14-15

O

- operator, referencing an entry, 15-28
- optimization
 - bit width example, 8-2
 - using various parameter types, 8-1
- outmulti hpeesoflang directive, 14-15, 14-26
- output hpeesoflang directive, 14-15, 14-25
- OutSDFPort class, 14-26
- outSDFport class, 14-24
- overflow
 - Fix class, 15-6

P

- parameter expressions, 4-8
- parameters
 - complex, 14-11
 - for fixed-point components, 4-9
 - with optimization attributes, 4-19
- ParamSweep component, 7-1
- Particle class, 15-19
- particle class, 14-24, 14-28
- particle types, 14-28
- performance optimization
 - See optimization, 8-1
- pl file template, 14-41
- polymorphism, 14-39
- porthole class, 14-24
- portholes
 - modifying in derived classes, 14-32
- porting
 - matrix stars, 16-3
 - message class, 16-2
 - Tcl/Tk issues, 16-1
 - UC Berkeley models, 16-1
- ports
 - hiding from the user, 14-32
- precision
 - parameter in HP Ptolemy, 15-5
- precision type
 - states, 14-10
- print method, 14-28
 - Message class, 15-24
- Printer (SDF block), 14-38
- private hpeesoflang directive, 14-18
- protected hpeesoflang directive, 14-18

public hpeesoflang directive, 14-18

Q

quantization

Fix class, 15-6

R

Ramp (SDF block), 14-36

reference count, 15-19, 15-24, 15-36

resistance, input/output, 9-9

resistors

cosimulation with clustering, 11-5

rounding

Fix class, 15-6

S

saturation

Fix class, 15-6

SDFFix class, 15-12

setAttributes method, 14-32

setAttributes method, 14-33

setInitValue method, 14-32

setSDFParams method, 14-16, 14-26

setup hpeesoflang directive, 14-17

sign bit, 15-6

sink models

examples, 14-51

writing, 14-45

state hpeesoflang directive, 14-28

states

hiding from the user, 14-32

modifying in derived classes, 14-32

string parameters, 4-17

string states, 14-10

stringarray states, 14-10

StringArrayState class, 14-29

StringState class, 14-29

Sweep Plan, 7-6

sweeping parameters, 7-1

performing multidimensional sweeps, 7-12

procedure, 7-2

using the VAR component, 7-4

using various parameter types, 7-6, 7-9

Synchronous Dataflow, 9-2

T

time converters

placing for cosimulation, 11-10

time step resolution, 9-8

time steps

size for cosimulation, 11-9

using for cosimulation, 11-9

timed components, writing, 14-33

Timed Synchronous Dataflow, 9-8

TkBarGraph, 12-9

TkBreakPt, 12-12

TkButtons, 12-11

TkMeter, 12-12

TkPlot, 12-3

TkShowBooleans, 12-12

TkShowValues, 12-6

TkSlider, 12-3

TkText, 12-6

TkXYPlot, 12-7

Transient simulator

time step size for cosimulation, 11-9

truncation

Fix class, 15-6

two's complement, 15-6

type conversion, 14-28

Message class, 15-23

U

underflow

Fix class, 15-6

uninitialized Fix object, 15-9

user-compiled signal processing models,

See model development, signal

processing

user-defined messages, 15-20

V

value types, 4-1

complex array, entering, 4-5

complex, entering, 4-2

entering, 4-1

enumerated type, entering, 4-6

filename, entering, 4-3

fixed point array, entering, 4-4

fixed point, entering, 4-2

integer array, entering, 4-3

integer, entering, 4-1

precision, entering, 4-2

real array, entering, 4-4

real, entering, 4-1

string array, entering, 4-6

- string, entering, 4-2
- table of, 4-1
- vector message, 15-20

W

- wrapup hpeesoflang directive, 14-18
- writing sink models, 14-45
 - writing data to dataset, 14-49