Agilent Technologies

Advanced Design System 2011.01

Feburary 2011
ADS Ptolemy Simulation

Acknowledgments

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

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

UMFPACK 5.0.2 - UMFPACK Notice: The UMFPACK code was modified. Used by permission. UMFPACK Copyright: UMFPACK Copyright © 1995-2006 by Timothy A. Davis. All Rights Reserved. UMFPACK License: Your use or distribution of UMFPACK or any modified version of UMFPACK implies that you agree to this License. This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code

and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. UMFPACK Availability: http://www.cise.ufl.edu/research/sparse/umfpack UMFPACK (including versions 2.2.1 and earlier, in FORTRAN) is available at http://www.cise.ufl.edu/research/sparse . MA38 is available in the Harwell Subroutine Library. This version of UMFPACK includes a modified form of COLAMD Version 2.0, originally released on Jan. 31, 2000, also available at http://www.cise.ufl.edu/research/sparse . COLAMD V2.0 is also incorporated as a built-in function in MATLAB version 6.1, by The MathWorks, Inc. http://www.mathworks.com . COLAMD V1.0 appears as a column-preordering in SuperLU (SuperLU is available at http://www.netlib.org). UMFPACK v4.0 is a built-in routine in MATLAB 6.5. UMFPACK v4.3 is a built-in routine in MATLAB 7.1.

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

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

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

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

these terms, the warranty terms in the separate agreement shall control.

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

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

About ADS Ptolemy

The ADS Ptolemy software provides 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. Cosimulation with ADS RF and analog simulators can be performed from the same schematic.

Using the ADS Ptolemy simulator you can:

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

And, ADS Ptolemy features:

- Timed synchronous dataflow simulation
- Easy-to-use interface for adding and sharing custom models
- Interface to test instruments
- Data display with post-processing capability

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

Built on the UC Berkeley Ptolemy code, ADS Ptolemy software includes a large number of behavioral, time-domain antenna and propagation models that are critical to communication systems designers. For DSP designers, fixed-point analysis is scalable up to 256 bits. The intuitive ADS user interface 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*. ADS Ptolemy has modified the proven synchronous dataflow domain to include timed components; this 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 *Theory of*

Operation for ADS Ptolemy Simulation (ptolemy).

Terminology

Throughout most of the ADS Ptolemy documentation, we use the ADS 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. The UC Berkeley Ptolemy terminology is used only in *Theory of Operation for ADS Ptolemy Simulation* (ptolemy) and in the topics on building signal processing models found in the *User-Defined Models* (modbuild) documentation.

Terminology Comparison

ADS Ptolemy Term	UC Berkeley Ptolemy Term
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 ADS 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 ADS 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 ADS 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 ADS 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

- A schema language used to define stars in ADS Ptolemy.
- 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 ADS 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 ADS 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.

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 ADS Ptolemy block diagram) from within the ADS 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 ADS Ptolemy. An atomic (indivisible) unit of computation in an ADS

Ptolemy application. Every ADS Ptolemy simulation ultimately consists of executing the methods of the stars used to define the simulation.

Star

The base class in the ADS 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 ADS 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 ADS 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 ADS 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 ADS Ptolemy are implemented in Tcl/Tk.

token

A unit of data in a dataflow model of computation. Tokens are implemented as particles in ADS Ptolemy.

universe

An entire ADS 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.

ADS Ptolemy GoldenGate Models

Using ADS Ptolemy, a system designer can create a DSP system design that is available to an Analog/RF designer as a source or a measurement model (Sink) in Agilent Technologies GoldenGate Simulator in Cadence Analog Design Environment. This mechanism enables a system designer to make complex signal generation and system measurements available to an Analog/RF designer. The Analog/RF designer can then validate and verify the Analog/RF design or device under test (DUT) using the GoldenGate Simulator.

This topic describes how to create and export ADS Ptolemy designs to GoldenGate.

Creating ADS Ptolemy Designs for Use in GoldenGate

The design must be created in a DSP Schematic window using only DSP components. A/RF cosimulation sub-circuits cannot be embedded inside the design. Any resistor(s) adjacent to the input or output port will have no impact on the Analog/RF DUT.

Currently, we only support exporting ADS Ptolemy source designs with all output ports and ADS Ptolemy sink designs with all input ports, separately.

ADS Ptolemy source design requirements for use in GoldenGate

- The ADS Ptolemy source design must not contain a DF controller.
- The ADS Ptolemy source design must not contain a sink type DSP component.
- The ADS Ptolemy source design must contain only output ports.
- The ARF_Export component (available under Circuit Cosimulation palette) must be used to specify additional port information for GoldenGate.
- The "*ComponentType*" parameter in ARF_Export must be set to "*SOURCE*"
- At most, one "*RF*" port is allowed in a source design to be exported to GoldenGate.
- The "*RFPortNum*" parameter in ARF_Export must be set to the RF port number if there is one or zero if there is no RF port in the schematic.
- The design should generate timed baseband and/or RF signal appropriate for stimulating the Analog/RF DUT.
- In the presence of an "*RF*" port, the top level source design must contain "*FCarrier*" and "*ROut*" as design parameters.
- The design parameters could be set using "*File->Design Parameters*" from the schematic window. These design parameters will be available to be modified in Cadence schematic capture.
- The frequency of signal at "*RF*" type port must always be equal to "*FCarrier*" defined as a design parameter. Please be careful if Analog/RF designer is expected to modify this parameter in Cadence environment while using GoldenGate simulator.
- Make sure the "*Not Edited*" and "*Not Netlisted*" are NOT checked for "*FCarrier*" and "*ROut*" design parameters.
- The source signal, in GoldenGate, from "*RF*" port will be at "*FCarrier*" Hz and will have "*ROut*" ohms output resistance.
- The top level design to be exported must contain a VAR block containing a variable named "*MaxTimeStep*", representing time step at the output port(s).
- The value of time step in GoldenGate simulator, set by Analog/RF designer, must be smaller than the "*MaxTimeStep*" when using the exported source.

ADS Ptolemy sink design requirements for use in GoldenGate

- The ADS Ptolemy sink design must contain only input ports.
- The ARF_Export component (available under Circuit Cosimulation palette) must be

used to specify additional port information for GoldenGate.

- The "*ComponentType*" parameter in ARF_Export must be set to "*SINK*"
- At most, one "*RF*" port is allowed in a sink design to be exported to GoldenGate.
- The "*RFPortNum*" parameter in ARF_Export must be set to the RF port number if there is one or zero if there is no RF port in the schematic.
- The design should read timed baseband and/or RF signal from Analog/RF DUT.
- In the presence of an "*RF*" port, the top level sink design must contain "*FMeasurement*" as design parameter.
- The design parameters could be set using "*File->Design Parameters*" from the schematic window. These design parameters will be available to be modified in Cadence schematic capture.
- The frequency of signal at "*RF*" type port will be equal to "*FMeasurement*" defined as a design parameter. Please be careful if Analog/RF designer is expected to modify this parameter in Cadence environment while using GoldenGate simulator.
- Make sure the "*Not Edited*" and "*Not Netlisted*" are NOT checked for "*FMeasurement* " design parameter.
- All ports in sink will exhibit infinite input resistance in GoldenGate. The output voltage at sink input in GoldenGate will depend on load resistance in GoldenGate Device Under Test (DUT).
- The top level design to be exported must contain a VAR block containing variable named "*MaxTimeStep*".
- The time step at all the input port(s) will be be adjusted to "*MaxTimeStep*" value automatically.
- The value of time step in GoldenGate simulator, set by Analog/RF designer, must be smaller than the "*MaxTimeStep*" when using the exported sink.
- The top level design to be exported must contain a VAR block containing variable named "*MinStopTime*", representing stop time for this sink. Th Stop time in GoldenGate simulator, set by Analog/RF designer, must be larger than the MinStopTime.

Each design can have parameters as required. These parameters will be available for modification in the Cadence environment while using the GoldenGate simulator. Parameters can be added using the *File > Design Parameters* in an ADS Schematic window.

Features Not Supported in ADS Ptolemy Design for Use in GoldenGate

The ADS Ptolemy Design to be exported must not contain any of the following at any level of design hierarchy

- Any Analog/RF sub-design as component.
- Any SystemC Cosim component.
- Any HDL Cosim component.
- Any Matlab Cosim component.
- Any Interactive Controls and Displays component.
- Any Instruments component.
- Any component that is reading a file, where file location is NOT specified using an absolute path.

Creating a Verification Testbench

Create a new top-level design and place the instances of the source and sink designs in it. Connect the source to the sink and add a DF controller. This top-level design must

simulate successfully before attempting to export Sink and Source designs to GoldenGate.

Exporting ADS Ptolemy Designs for Use in GoldenGate

In ADS 2011 ARF_ExportPort has been removed and its functionality has been replaced with the new ARF_Export controller. The automatic migration process in ADS will remove the instances of ARF_ExportPort from GoldenGate source or sink deigns and insert the ARF_Export controller. However, there is a known issue with the RFPortNum parameter. In some designs with baseband ports the parameter may be set incorrectly. The user will need to set the RFPortNum parameter to the RF port number, or to zero if there are only baseband ports in the design. If the parameter does not match the actual RF port number, the GoldenGate simulation will fail with an error "FCarrier specified is not similar to the carrier frequency of RF signal in Ptolemy," because GoldenGate treats baseband and RF ports differently.

Export GoldenGate Virtual Testbench

You can export either the whole verification testbench as a Virtual Testbench or the source or sink as a Virtual Testbench component. Select *Tools > Export ADS Ptolemy Design > As GoldenGate VTB* in the schematic window containing the design. You will be prompted for the location of the output files, and an *Export Status/Error/Warning* dialog box will appear showing the export progress.

A successful export process generates the files *<Design name>.vtb*, and *<Design name>_GG.net*.

Export GoldenGate Model

To export the CDF file, save the source or sink design and select *Tools > Export ADS Ptolemy Design > As GoldenGate model* in the schematic window containing the design (you will need to push into the source or sink from the verification testbench). An *Export Status/Error/Warning* dialog box appears showing the exporting progress. A temporary design window will open where the design will be instantiated before the export starts.

A successful export process generates the files *<Design name>_GG.cdf*, and *<Design name>_GG.net*. These files are saved with the exported design under the workspace directory *..._wrk/adsptolemy/GoldenGate*.

Creating a Results Display for ADS Ptolemy Designs Used in GoldenGate

An ADS Ptolemy design may have complex data that an Analog/RF designer will not know how to interpret. To help simplify data analysis, the System designer must use the following steps:

1. System designer must perform verification of the design as mentioned earlier to create a dataset in ADS.
2. Open a new DDS window. Add one or more new pages and name them appropriately.
3. Add correct equations/plots/configurations on the new pages.
4. Save this DDS file as a template by choosing *File > Save as Template* on the DDS window. Select the *User* category to save the template.
5. The template file is saved under *$HOME/hpeesof/circuit/templates*.

6. Copy the template file to ..._*wrk/adsptolemy/templates*.
7. Place an OutputOption controller in the designs schematic window. (The OutputOption component can be placed at any level of hierarchy in the design.)
8. Add the data display template name, created above, to the list of names. More than one data display template is allowed for a given design.
9. Re-export the design.

When this design is used in a GoldenGate simulation, the templates listed in the OutputOption controller will be inserted in the DDS window. The users can launch DDS via *Results > Start ADS DDS Display ...* menu item in ADE. Please consult GoldenGate documentation for further details about invoking Data Display inside GoldenGate-ADE.

Importing ADS Ptolemy Design to Use in GoldenGate in Cadence Environment

If you exported a Virtual Test Bench, you can use *Setup > Virtual Test Bench...* in *Analog Environment* to set it up.

The ADS Ptolemy Designs exported as CDF files can be imported using *Tools > Import ADS Ptolemy Models into GoldenGate...*.

Please consult GoldenGate documentation for further details about import process and how to simulate these models.

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. ADS Ptolemy provides the signal processing simulation, while the analog/RF simulation is provided by either the Circuit Envelope or High-Frequency SPICE (Transient) simulators.

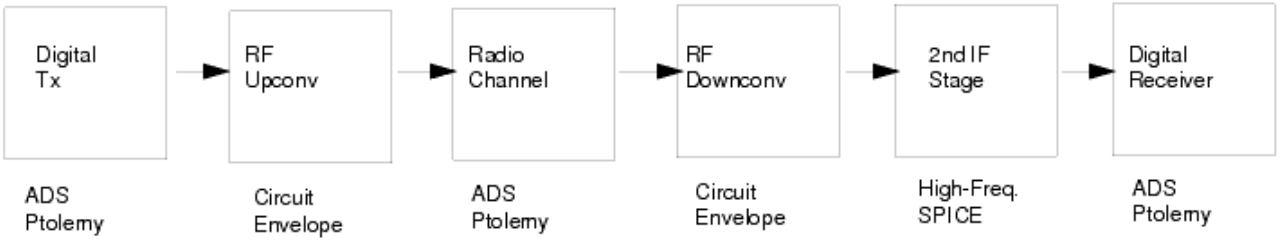
Other types of cosimulation include placing MATLAB components or HDL blocks in a signal processing simulation. This topic describes cosimulation with analog/RF systems.

Note
For information on A/RF cosimulation with Cadence refer to the *Cadence Library Integration* (dynlnklc) documentation. For specific details, see *Running a DSP and Analog - RF Cosimulation with RFIC Dynamic Link* (dynlnkug).

The following figure shows a mixture of RF circuitry and DSP components. ADS provides a variety of analog/RF circuit simulators, including Linear, Harmonic Balance, Circuit Envelope, High-Frequency SPICE, and Convolution.

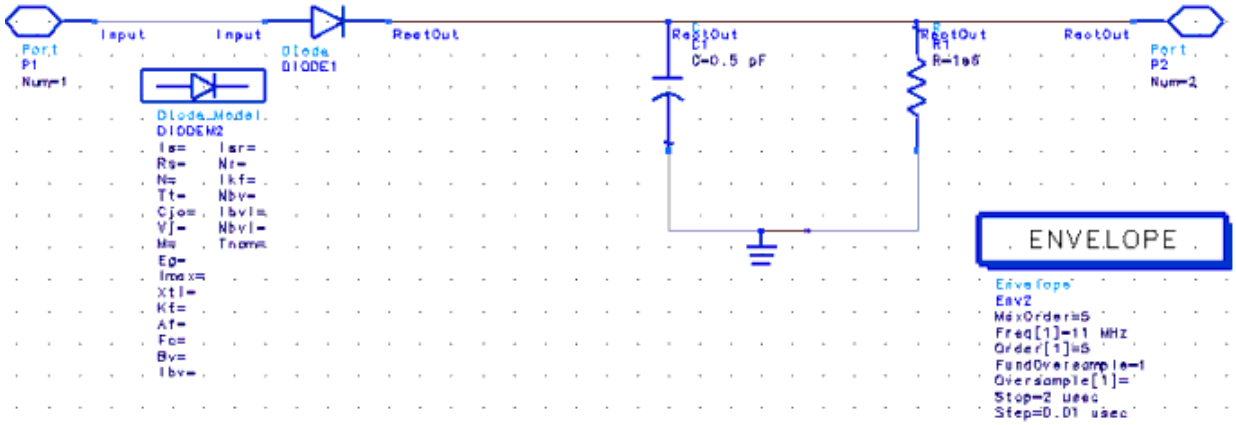
Note
Circuit Envelope and High-Frequency SPICE simulators are included with some, not all, ADS suites.

For signal processing simulation, ADS 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 described in the next section.



Cosimulation: Different Design Portions Simulated by Different Simulators in the Same Schematic

Setting Up the Analog/RF Circuit Schematic



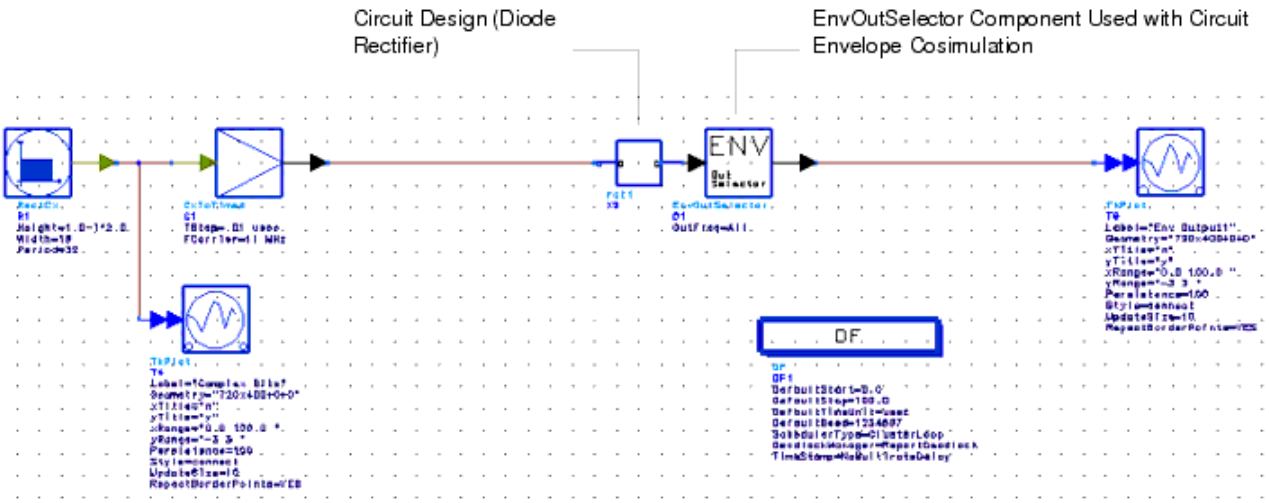
Diode Rectifier Circuit Design Used in Cosimulation

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 Envelope and Transient simulators in the same design; if you want to keep both controllers in a design use activate/deactivate.
4. Add ports to your design.
5. Save your design.

In the previous 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



Signal Processing Design Using Circuit Shown in Previous Figure

To create signal processing designs for cosimulation:

1. Choose the circuit design you want and place it in your signal processing schematic.
2. Add the signal processing components.
3. Add the signal processing controller(s).
4. Connect the circuit design to the signal processing components.
5. For cosimulation with the Circuit Envelope simulator, see Circuit Simulation Controllers.
6. If your circuit subnetworks have feedback loops between them, see Feedback Loops.
7. If the input signal into the circuit subnetwork is not of type Timed, see Numeric-to-Timed Converters.
8. Select the initialization method of the circuit inputs in the Signal Processing DF controller; see *DF (Data Flow) Controller* (ptolemy).
9. Start the simulation.

Circuit Simulation Controllers

As stated earlier, ADS 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 must be the *timed* type. If the input component (connecting signal processing components to the circuit) produces numeric data, place an appropriate numeric-to-timed converter (such as float-to-timed or complex-to-timed) in your schematic. These components (located in the Signal Converters library) ensure that the input into the circuit subnetwork is in the time domain. Refer to Time Converters for more information.

Automatic Verification Modeling (Fast Cosimulation)

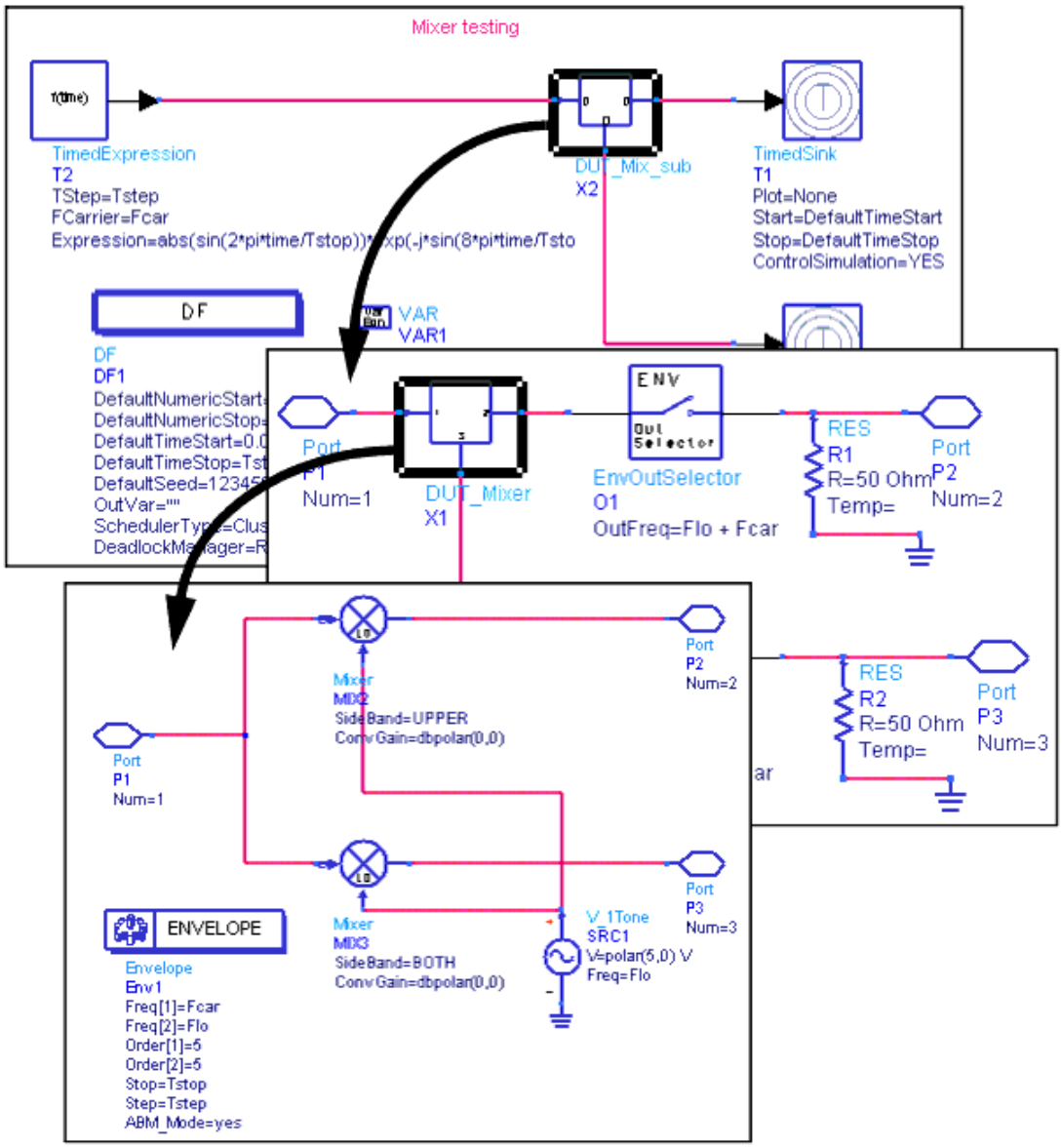
Automatic Verification Modeling is a simulation mode that can significantly accelerate formerly lengthy cosimulations of Analog/RF circuits. You can enable Automatic Verification Modeling in the Circuit Envelope Simulation Controller. When enabled, this mode characterizes an analog subcircuit into a behavior model, then the model is used to predict the response of the subcircuit at each time point. For details about Automatic Verification Modeling, see *Automatic Verification Modeling* (cktsimenv).

The following steps demonstrate how to enable the fast cosimulation mode using *PtolemyDocExamples/AVM_wrk* from the examples directory:

1. Select *AVM_wrk* from the *PtolemyDocExamples* directory.
2. Open the design *TestMixer*.
3. In the Schematic window, select DUT_Mix_sub, and push into its hierarchy, then push into the DUT_Mixer hierarchy as shown in the following figure.
4. In the DUT_Mixer, double-click the Envelope simulation controller to open its setup

dialog. Select the Cosim tab, and click **Enable AVM (Fast Cosim)** to enable the mode.

> **Note**
> To enable AVM (Fast Cosim) directly on the schematic, click the Display tab on the Circuit Envelope setup dialog. Enable the **ABM_Mode** parameter. On the schematic, set **ABM_Mode=yes** to enable the mode; set **ABM_Mode=no** to disable the mode.



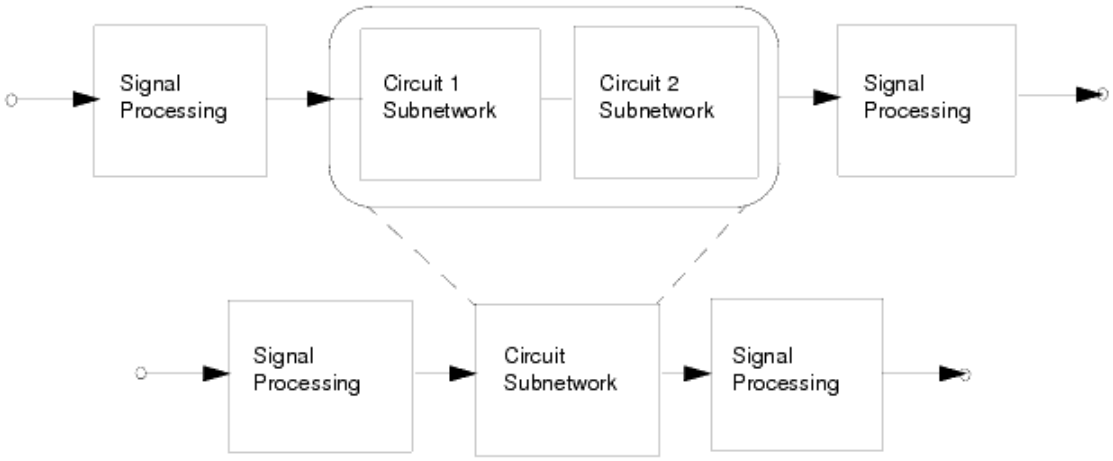
Pushing into *TestMixer* Hierarchy to Enable AVM (Fast Cosim)

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 and treated as one circuit subnetwork, as shown in the following

figure. Therefore, use only one circuit simulation control component in either of the two (or more) directly connected subnetworks.



Connected Subnetworks Treated as One

Connected Circuit Subnetworks

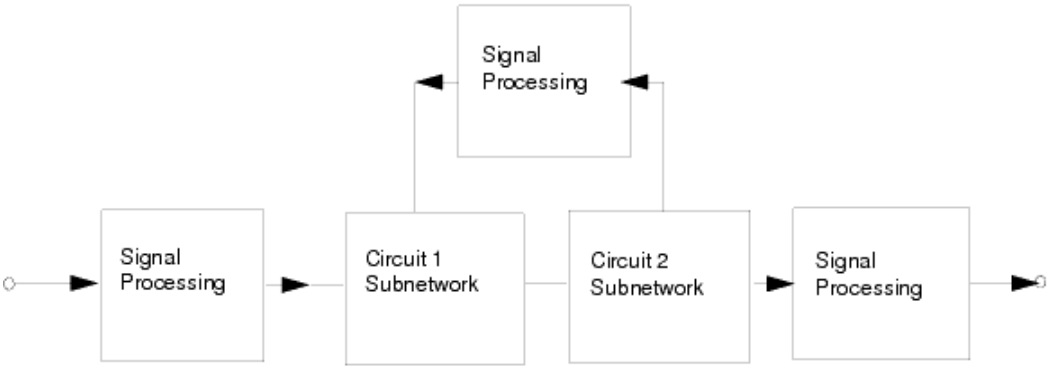
When two circuit subnetworks defined on two different circuit schematics are connected on a Signal Processing schematic, the two circuit subnetworks are clustered into one (this is done transparently and should not concern the user). However, if each of these two circuit subnetworks use their own simulation controller, then the circuit engine would not know which one to choose for simulation and would result in an error message.

Connected Resistors

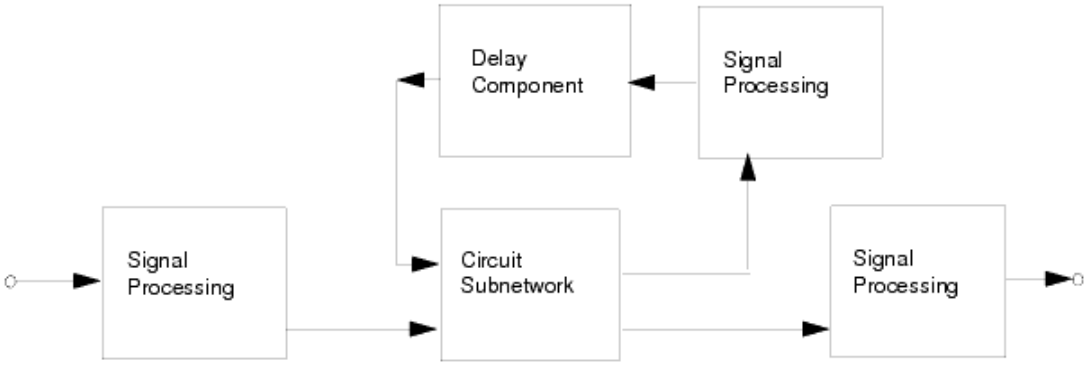
Another aspect of clustering is when circuit components available on the Signal Processing schematic (resistors in the first release of Advanced Design System) are connected to a circuit subnetwork. In this case, such resistors will be absorbed into the circuit subnetwork during the clustering and will 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 the following two figures. If such a delay is not present, an error message will be issued. To have the program automatically insert the delay, you must edit the DF (data flow) controller parameters. To do this, double-click the controller, choose the *Options* tab, then select *Resolve deadlock by inserting tokens* from the DeadlockManager drop-down list. For more information about deadlocks, refer to *Deadlocks* (ptolemy).



Feedback Loop Before Delay Added by Program



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 must 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 for more information. 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's *Circuit Envelope*

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* or *EnvOutShort* component is among the fundamental frequency or harmonics specified by the Circuit Envelope controller.

To enable or disable the addition of noise from the Circuit Envelope components, use the *Turn on all noise* parameter in the Env Params tab. For cosimulation, the *Nonlinear* noise button, the *Small-signal* button, and their corresponding tab pages should not be used as they are not used during cosimulation. Note that explicit noise sources, such as _V_Noise_, are always on. The *Turn on all noise* enables or disables only the noise generated by non-source models, such as the resistors, lossy transmission lines, and transistors.

The amplitude of these noise sources is determined by the Circuit Envelope bandwidth, which is determined by 1/(Time Step). Normally, this Circuit Envelope timestep would be the same timestep as that used by Ptolemy, so the total noise bandwidths are the same. However, for designs where the Ptolemy waveform is changing too rapidly for the analog simulation and the user has reduced the Circuit Envelope timestep, then the Circuit Envelope noise bandwidth also increases. Depending on the circuit behavior, this broader bandwidth noise may appear at the circuit output, where it is now effectively sampled by the EnvOut element. This sampling will then alias all the higher bandwidth circuit noise, and result in a higher noise density within the Ptolemy noise bandwidth. If this is not the desired simulation behavior, then a filter may need to be placed at the circuit output. Of course, this will also filter any signals being passed back to the Ptolemy simulation.

Transient Simulation Specific Rules

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

There is a difference between ADS Ptolemy and the Transient simulator regarding how they treat signals at time=0- (before t=0) that may cause unexpected results; they have different simulation assumptions for time=0-. ADS Ptolemy assumes signal states at time=0- are zeros, while the Transient simulator assumes signal states at time=0- the same as time=0. For a circuit with a given input signal stimulus to give the same response for both Ptolemy/Transient cosimulation and for Transient simulation alone, the circuit signal stimulus for the Ptolemy/Transient cosimulation must have a zero value at time=0. If the signal stimulus into the circuit during a Ptolemy/Transient cosimulation is not zero at time=0, then the cosimulated Transient simulator will result in output signals that are not expected when compared to a Transient-alone simulation. To force the circuit input to be zero during a Ptolemy/Transient cosimulation, the workaround is to change the CktCosimInputs setting on the Options tab of the DF (data flow) controller (refer to *Options Tab* (ptolemy) for details).

Nested Simulation Approach

ADS 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

ADS Ptolemy uses a data flow simulation approach, and this simulation is controlled using the DF (data flow) controller component. To understand this section, 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 completes 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 circuit that is cosimulated with the signal processing design is the clustered circuit subnetwork (see Clustering of Circuit Subnetworks) inclusive of any resistor (RES) component associated with signal processing components at the inputs or outputs of the circuit. Thus the circuit is interfaced to signal processing components that appear as ideal sources at the circuit inputs (with 0 ohm source resistance) and as ideal loads at the circuit outputs (with infinite ohm load resistance). The circuit output can interface directly to signal processing components at the circuit outputs for circuit transient cosimulation. However, for circuit envelope cosimulation, the circuit outputs must be connected to an envelope selector component (EnvOutSelector, EnvOutShort) which is then connected to signal processing components. The EnvOutSelector output provides an open circuit to any resistor associated with an attached signal processing component input. The EnvOutShort output provides connectivity to any resistor associated with an attached signal processing component input. Each circuit input may have only one zero ohm signal processing source connected. When multiple signal processing signals are desired at one circuit input, then each zero ohm source can have their signals combined with any suitable resistor combining network. Those source resistors are typically included in the definition of *Timed* signal processing

component outputs. The user must keep in mind that any resistor (RES) component associated with signal processing components at the inputs or outputs of the circuit are really part of the circuit and not part of the 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 *Transient and Convolution Simulation* (cktsimtrans).

For cosimulation with the Transient simulator, keep in mind one key issue: The Transient simulator may need time steps smaller than ADS 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 ADS Ptolemy.

Note
For all practical purposes, the only parameter that may concern the cosimulation user is the Max time step. Other parameters in the Transient Simulation control component can remain at default values.
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.

Depending on the Time Step value you set, the simulator will set the internal Circuit Envelope time step to either the Ptolemy time step value, or to an integer sub-multiple of this value. Time step values less than the Ptolemy time step value are sometimes required to achieve the desired accuracy, due to rapidly changing signals and the integration required for capacitive and inductive components. If the Circuit Envelope value is less than one-tenth of the Ptolemy time step value, a warning is generated to alert you to this so you can avoid inadvertent small time step values that might unnecessarily be slowing the cosimulation.

Note
The Stop time for the simulation is determined by the Signal Processing Data Flow controller and/or Sinks. The Circuit Envelope Stop time does not affect the duration of the cosimulation. The Stop Time value is used in a few models and, ideally, should be set to reflect an approximate stop time range. As an example, for explicit Analog/RF Noise sources that have a user-specified baseband frequency response, this stop time value is used to determine the maximum duration of the pre-computed random noise sequence. Presently, the stop time is limited to about 2.1M (=2^{21}) times the time step value. If the cosimulation runs longer than this, then this noise data is repeated.

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 DelayRF component in such feedback loops will suffice. If such a delay does not exist, ADS 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, FixToTimed, IntToTimed, or CxToTimed converter to accommodate the conversion. Although ADS 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 may 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 ADS 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.

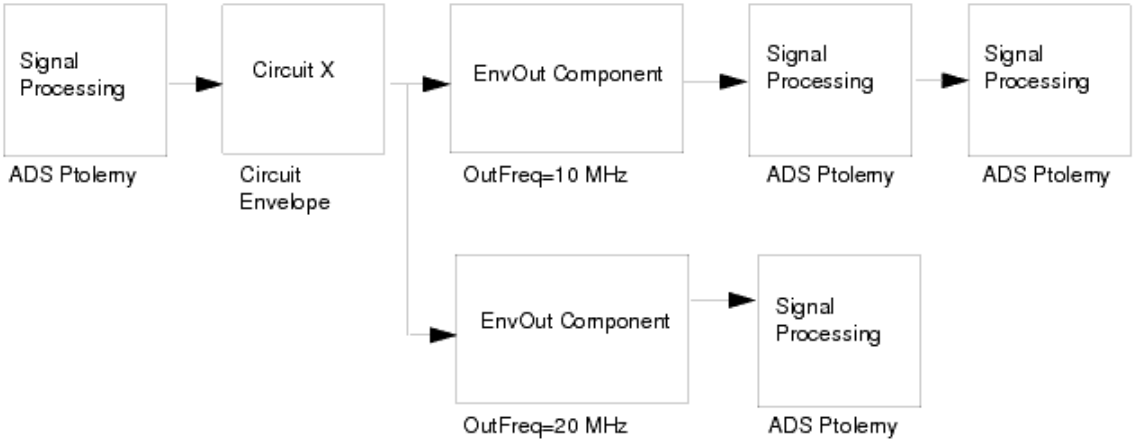
The EnvOutSelector component acts as an open, blocking everything connected to its output from loading the circuit. If such loading is desired, use the EnvOutShort

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 EnvOut components (EnvOutSelector and EnvOutShort) can be connected to each output port of a circuit subnetwork as illustrated in the following figure. All waveforms generated by the Circuit Envelope simulator can be accessed in a Signal Processing schematic.



***EnvOut* Components (*EnvOutSelector* or *EnvOutShort*) at each Circuit Subnetwork Output Port**

When an EnvOutSelector or EnvOutShort component is used with a design simulated by the Transient simulator, their effect is an open or a short, respectively. Otherwise they do not affect transient cosimulation designs and can remain in place without any impact on the cosimulation.

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. 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

While the cosimulation use model is intuitive, the following information will help you avoid

errors.

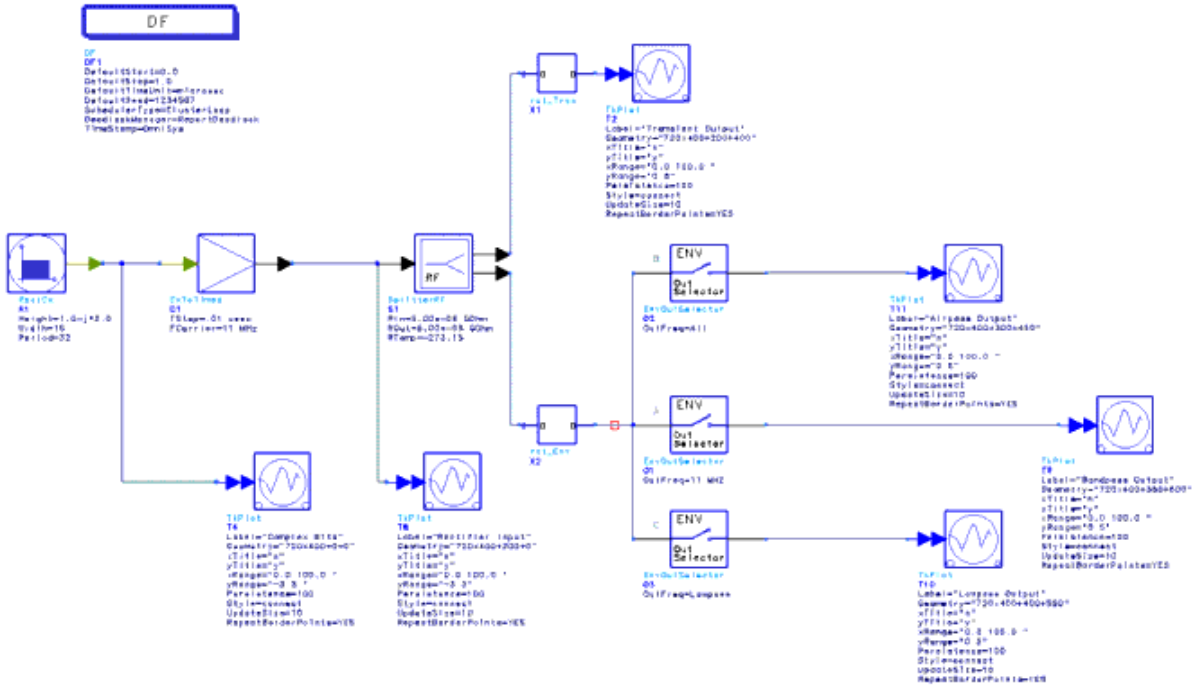
1. Only the Transient and Circuit Envelope circuit simulators can cosimulate with ADS Ptolemy. Other circuit simulation controllers on the analog/RF schematic (such as S-parameter or AC) will be ignored in cosimulation.
2. Directly connected circuit subnetworks placed as instances on Signal Processing schematics are clustered together and 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.
5. Writing a VAR in the analog/RF subnetwork to the dataset *cannot* be done using the Output tab on the Circuit Envelope Controller's setup dialog box, nor by using OutVar in the Data Flow Controller. It can be done by using the Output tab option in the top-level DF controller, or defining the VAR in the top-level DSP design.

Cosimulation Example

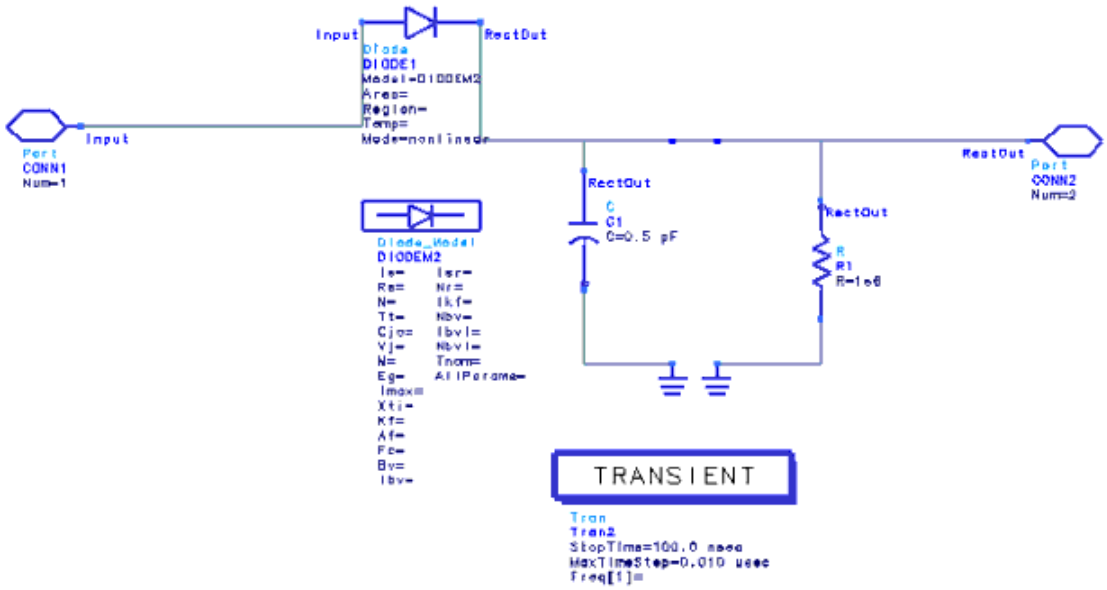
To illustrate cosimulation, we will use an example called RectifierCosim_wrk from the */Com_Sys* directory.

The top level design RectifierCx_Tutorial, as shown in the immediately following figure, generates a complex modulated signal and sends that signal into a simple rectifier circuit.

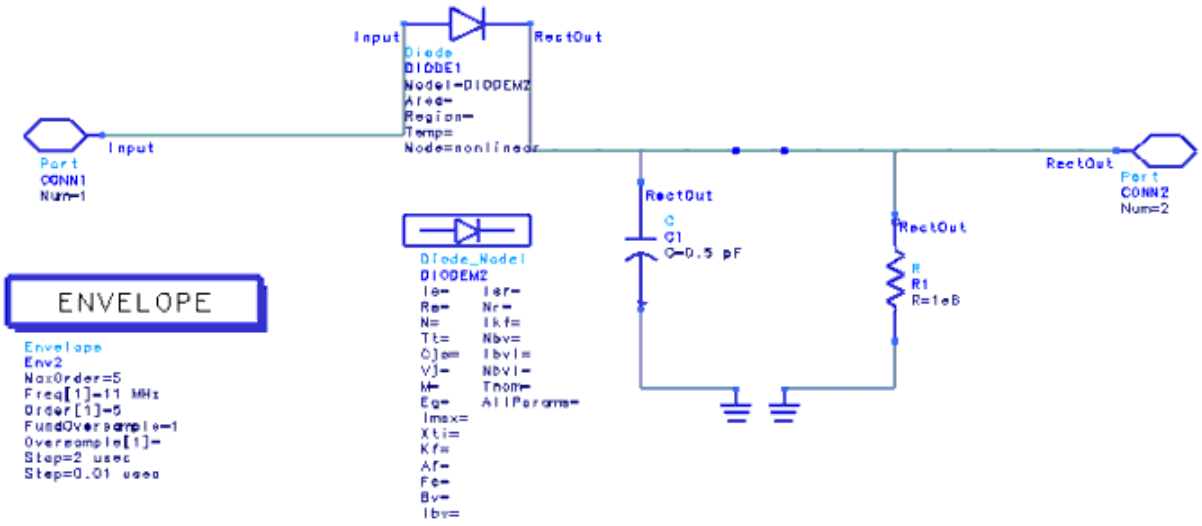
Two identical instances of this rectifier circuit are created on a circuit schematic, one with a TRAN controller (rct_Tran), in the second figure below, and the other with an ENV controller (rct_Env), in the third figure below.



RectifierCosim Top-Level Schematic



Circuit Subnetwork with Transient Controller



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 µsec 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.

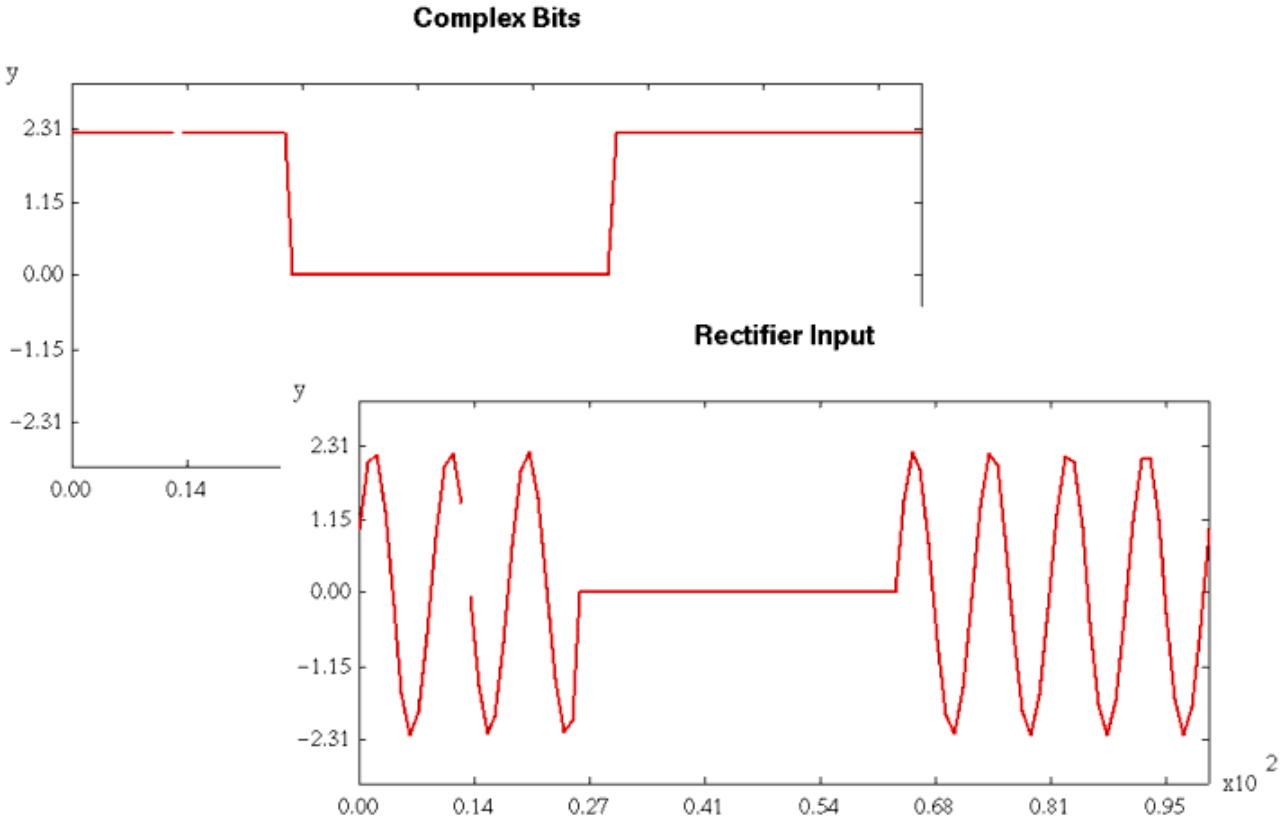
Since the TStep of the signal entering circuit design is at 0.01 µsec, the MaxTimeStep on the Transient controller is set to the same value. This value should always be smaller than or equal to the Signal Processing TStep. The other Time setup parameters, such as Start time and Stop time, are ignored 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.

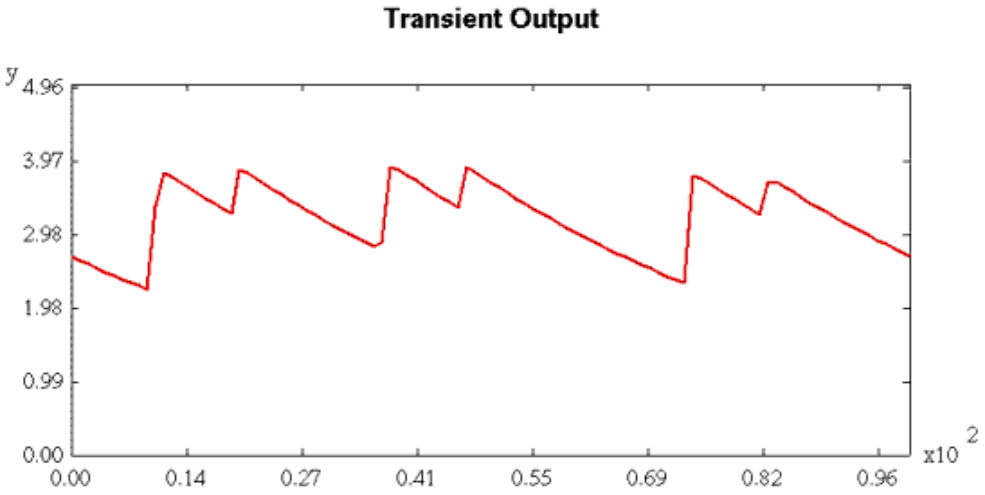
Typically, there is only one EnvOut component (*EnvOutSelector* or *EnvOutShort*) attached to the Circuit Envelope subnetwork output, but in this example we have used three to show the different signals that can be selected from the Circuit Envelope output. Specifically, the OutFreq parameter is set to the Bandpass, Allpass and Lowpass options in the three instances. When Bandpass 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, 6 TkPlot windows pop up, as shown next.

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.

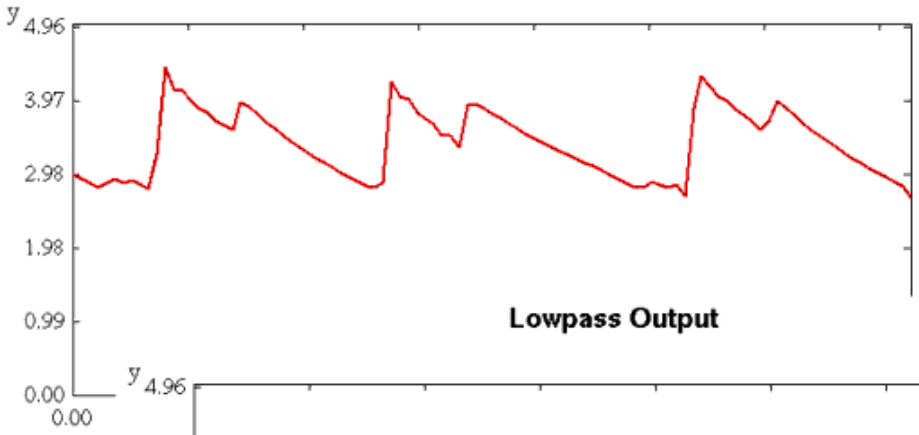


The plot *Transient Output* is the rectified version of the modulated signal (note that there are no negative components in the signal), where the value of the time constant (RC) determines the degree of pulse fall-off. Note also that this output is a real-baseband signal and includes all the harmonics.

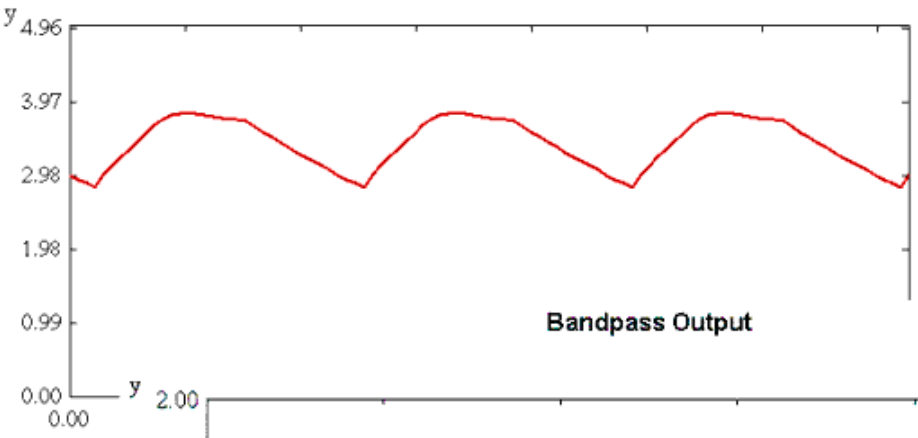


The allpass, lowpass, and bandpass output plots are shown next.

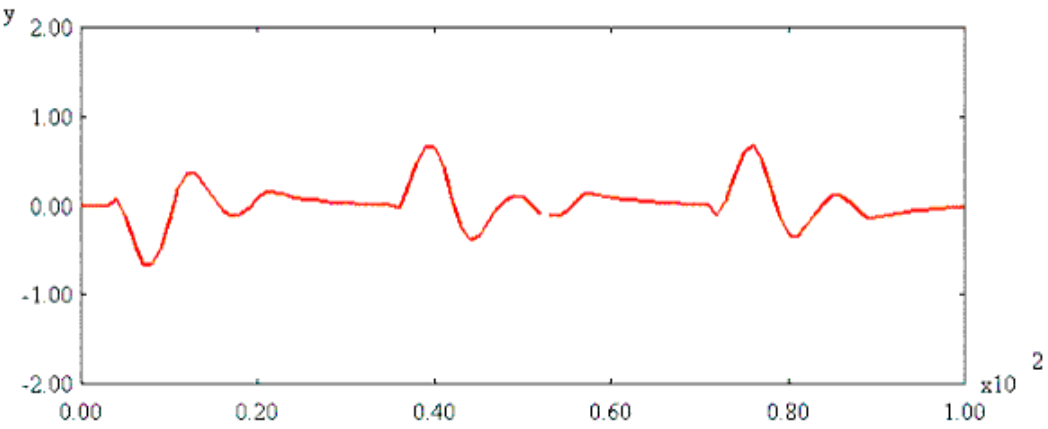
Allpass Output



Lowpass Output



Bandpass Output



Data Types, Controllers, Sinks, and Components

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

Representation of Data Types

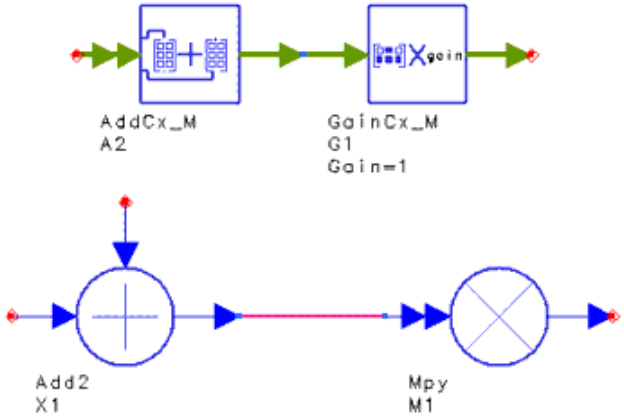
ADS Ptolemy components have stems of different colors and thicknesses that are based on the *data type* (this differs from Analog/RF Systems components). The following table lists the data types.

> **Note**
> 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.

Data Type Representation

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

The following figure shows the thicker stem width associated with matrix data compared to the thinner stem width associated with scalar data.



Matrix Data (Thick Lines) Versus Scalar Data (Thin Lines)

Components have single or multiple arrowheads at inputs or outputs.

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

For example, a multiple input multiplier component has multiple arrowheads at the input and a single output arrowhead, as shown in the previous figure.

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), data is copied from one component to another. If you connect components represented by different data types, such as scalar complex to scalar floating-point (real), or scalar integer to matrix integer, 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?

Although the software will automatically convert dissimilar data types, such as complex to floating-point (real), 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.

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

- When either a Float (real) 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 must define the time step at least once in your design.
- When a Complex port is connected to a Timed port. *Automatic* conversion from Complex to Timed is not supported. You must place a Complex to Timed converter between the ports and enter appropriate parameters.

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 floating-point (real), no data is lost; only the format is changed.

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

- Complex to Float (Real) ADS Ptolemy calculates the magnitude and ignores the phase.
- Complex to Fixed After calculating the floating-point (real) magnitude, ADS Ptolemy converts the floating-point (real) to fixed.
- Complex to Integer After calculating the floating-point (real) magnitude, ADS Ptolemy converts the floating-point (real) 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, Fc, time, plus an ADS 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 (Real) or Float (Real) 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* (ptolemy).

Controllers

Controllers, used to control simulation, are placed unconnected any where in the schematic, and are found in the Controllers library or palette. The DF (data flow) controller controls the flow of mixed numeric and timed signals for all digital signal processing simulations within ADS. 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*.

DF (Data Flow) Controller

The DF (data flow) controller is required for all simulations. Use the DF controller to

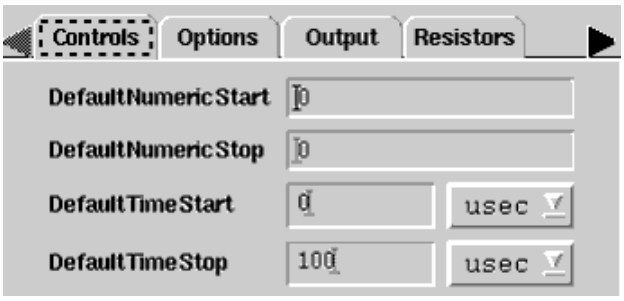
control the flow of mixed numeric and timed signals for all digital signal processing simulations within ADS. This controller, together with source and sink components, provide the flexibility to control the duration of simulation globally or locally.

Important
Multiple DF controllers on the schematic are not allowed.

Versions of ADS Ptolemy released before ADS 1.5 allowed multiple DF controllers on the same schematic. Starting with ADS 1.5, this is no longer possible. Multiple controllers were used to simulate the same design with different DF 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.

The DF controller dialog box has the Controls, Options, Output, Resistors, Debug, and Display tabs, which are described in the following sections.

Controls Tab



ADS 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 data 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.

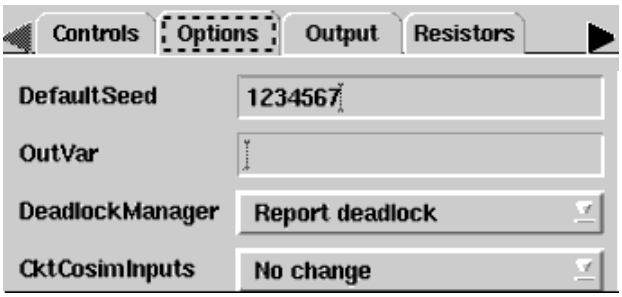
In timed sinks, Start and Stop have timed units because the data has a time base. The amount of data that the sink collects is a function of both the data time base and the sink's Start and Stop parameters. For example, if Start=0 µsec, Stop=1 µsec, and the data has a time base of 2 µsec, the sink will 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 **DefaultTimeStart** and **DefaultTimeStop**. Default values for these DF controller values are 0, 100, 0 µsec, and 100 µsec, respectively.

Sinks can control simulation locally with their own start and stop times, or they can use the appropriate DF 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 DF 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 options tab has the following parameters:

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

DefaultSeed 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 predictable simulation results.

To generate repeatable *random* output from simulation to simulation, use any positive seed value. For the output to be truly random, enter a seed value of 0.

OutVar

Note
OutVar is an obsolete parameter. Use the Output tab (refer to Output Tab) and the OutputOption controller (refer to OutputOption Controller).

OutVar is a space-separated list of variable names defined using variables and equations (VAR) components. Values are 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 places a set of quotes around the OutVar variable. Do not enter your own quotes as the double set will cause simulation failure.

The global character * is no longer supported.

DeadlockManager The Deadlock Manager enables you to manage design deadlocks. A deadlock 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.

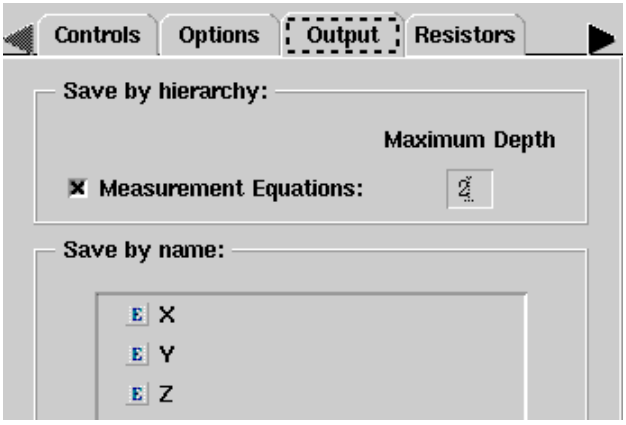
Select the type of deadlock management from the drop-down list:

- *Report deadlock* (default) Indicates the design includes deadlocks.
- *Identify deadlocked loop* s Enables you to spot which loops are deadlocked. These loops can be highlighted on the schematic page by clicking on the error message or Status window.
- *Resolve deadlock by inserting tokens* Adds delays to deadlock loops and allows the simulator to proceed.

CktCosimInputs This option controls the initialization method on the input pins of Analog/RF circuits during cosimulation. The option applies initialization to all cosimulation circuit subnetworks. (Refer to *Cosimulation with Analog-RF Systems* (ptolemy) for information regarding how to set up an ADS Ptolemy A/RF cosimulation.) Select the type of input initialization method from the drop-down list:

- *No change* (default) No special initialization.
- *Initialize with zero volts* Initialize the first data of all input pins to 0; basically, the first data value is discarded.
- *Insert one time step delay* Insert one extra data with 0 value to all input pins, delaying everything by one time step.

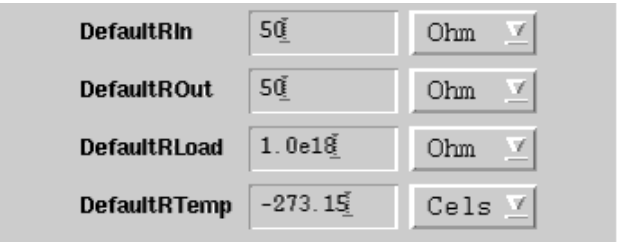
Output Tab



The Output tab enables you to selectively save simulation data to a dataset. For details, refer to *Selectively Saving and Controlling Simulation Data* (cktsim).

Note
Node Voltages are supported only on A/RF controllers; therefore, you will not find this option available on the Data Flow Controller's Output tab.

Resistors Tab



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 only.

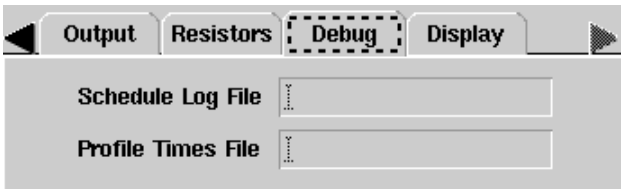
DefaultRIn is the default input impedance of timed components; its value is 50 ohms.

DefaultROut is the default output impedance of timed components; its value is 50 ohms.

DefaultRLoad is the default input impedance of timed sinks and the default impedance of solitary resistors (the R component); its value is 1.0e18 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

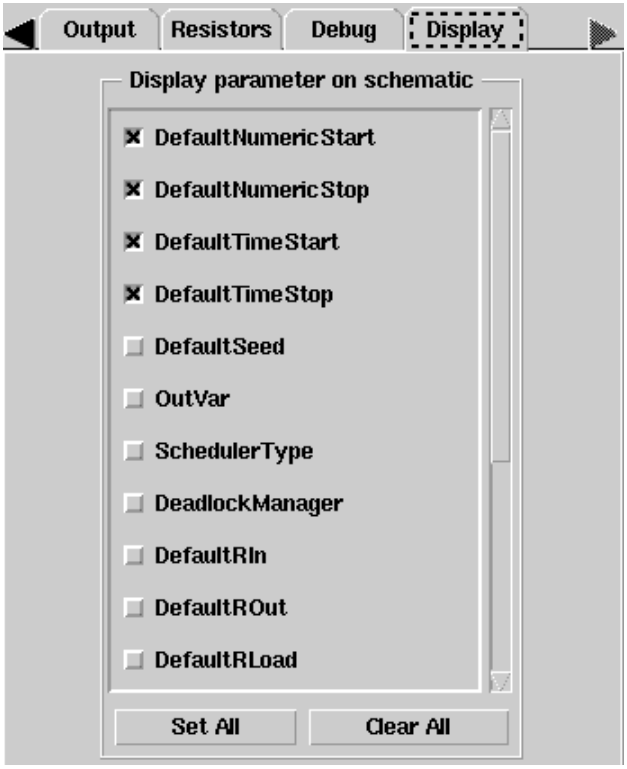


The Debug tab can be used to provide the ability to debug your design and its custom components.

Schedule Log File enables you to specify the file name for a log file. After simulation, the log file you specified will be generated under the / *data* directory of the workspace. It will log the firing schedule of components in your design.

Profile Times File enables you to specify the file name for a file containing simulation information. After simulation, the file will be located under the / *data* directory of the workspace. It provides run-time information for components in your design during simulation. For example, information may include the number of times a component is fired or the average time.

Display Tab



From the Display tab, you can choose which parameters to display on the schematic. By default, only the start and stop parameters are selected; choose which parameters to display by selecting or unselecting parameters.

OutputOption Controller

Description: Output Option for Dataset Templates
Library: Controllers

Parameters

Name	Description	Default	Type	Range
DatasetTemplate	Dataset Template name (repeatable)		string	

Notes/Equations

1. The OutputOption controller is used to specify the data display template(s) for Wireless Test Bench (WTB).
2. All listed data display template(s) will be automatically inserted into a data display window after simulation.
3. The string value of *DatasetTemplate* should be the name of only one data display

template file and should not contain the data display template file extension (.ddt).
4. To list more than one template name, use the *Add* button on the component dialog box to add additional *DatasetTemplate* parameters, each of which should have a string value for only one template.
5. A blank space (" ") value for *DatasetTemplate* will be ignored.

VarEqnCheck

Description: Check value range from a variable or expression
Library: Controllers

Parameters

Name	Description	Default	Type	Range
ParameterRange	Boolean expression for the parameter range check (value of 1 means ParameterValue within range limits)	X>=1	int	
ParameterValue	Optional value that can be reported when a message is to be displayed	X	real	
ReportValue	Report parameter value	NO	query	NO, YES
Type	Range check message type	Warning	enum	Warning, Status message, Initialization error, Abort simulation error, Standard
Message	Message to display if ParmeterRange evaluates to 0	`X is out of range, it should be >=1'	string	

Notes/Equations

1. Use VarEqnCheck to check the value range limits of a parameter defined either in a VAR item (variable or expression) or defined as a subnetwork parameter.
2. *ParameterRange* performs parameter value range test on the value defined by *ParameterValue*. *ParameterRange* should be a boolean expression that evaluates to 1 when the range test is successful. A range test is considered to be a failure when *ParameterRange* is zero and results in display of the message defined by *Message*.
3. *ReportValue* , when set to *YES* , results in display of the *ParameterValue* along with the *Message* when the *ParameterRange* is zero.
4. *Type* is the type of message displayed.
5. When *Type* is *Warning* , then the range test is performed during simulation. A range test failure (*ParameterRange* =0) results in display of the *Message* in the Simulation

Message window (and optionally *ParameterValue* when *ReportValue* = *YES*) and continuance of the simulation.

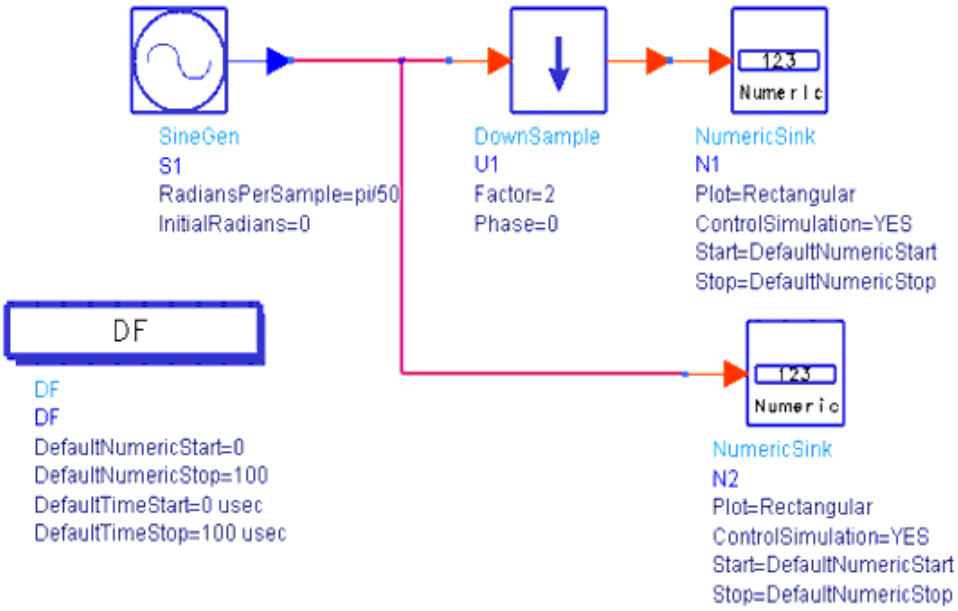
6. When *Type* is *Status message* , then the range test is performed during simulation. A range test failure (*ParameterRange* =0) results in display of the *Message* in the Simulation Status window (and optionally *ParameterValue* when *ReportValue* = *YES*) and continuance of the simulation.
7. When *Type* is *Initialization error* , then the range test is performed during simulation initialization before simulation begins. A range test failure (*ParameterRange* =0) results in display of the *Message* in the Simulation Message window (and optionally *ParameterValue* when *ReportValue* = *YES*). The simulation will continue for the remainder of the simulation initilization and will quit after all initialization errors are detected.
8. When *Type* is *Abort simulation error* , then the range test is performed during simulation. A range test failure (*ParameterRange* =0) results in display of the *Message* in the Simulation Message window (and optionally *ParameterValue* when *ReportValue* = *YES*) and simulation will quit.

Sources and Sinks Control the Simulation

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

Sinks

Sinks are components with no outputs. When a sink controls the simulation (ControlSimulation=YES), data collection always begins at the input sample with an index value equal to the value of the Start time parameter. The data collection ends with an index value equal to the value of the Stop time parameter. (One or both of the Start and Stop times might be inherited from the Data Flow controller.) By default, a sink's ControlSimulation parameter is set to YES. When a sink is not controlling the simulation (ControlSimulation=NO), it will start collecting data at Start, then collect as much data as the simulation produces. Consider the following example:



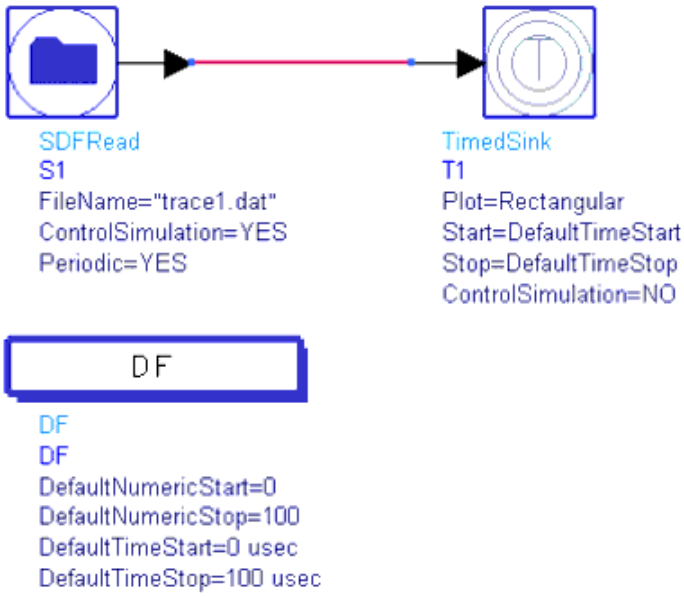
As shown, both sinks will collect 101 data samples (0 to 100 inclusive). They are both controlling sinks so they will obey their start and stop index times. Because of the DownSample, sink N2 will receive more data but it will not collect it.

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 N1 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 datasets 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 next.



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.

ADS Ptolemy Components

The component libraries available for use with signal processing designs using the ADS Ptolemy simulator are listed in the following table. 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.

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

ADS Ptolemy Component Libraries

Library	Summary of Contents
Antennas & Propagation	Components for radio channel, including antennas and propagation models. The channel models provide built-in functionality based on various standards: 1xEV, 3GPP, DTV, GSM, TDSCDMA, CDMA, WLAN.
Circuit Cosimulation	Items used to set up cosimulation with analog/RF circuits.
Common Components	A factory list of the most commonly-used components.
Controllers	Items that control simulation parameters.
HDL Blocks	HDL cosimulation components.
Instruments	Components used to link data to instruments, such as the 89600 Vector Signal Analyzer.
Interactive Controls and Displays	Components that control and interactively display real-time simulation results. Data is not saved.
Numeric Advanced Comm	Components that provide functions for simulation of advanced communication systems based on the latest communication technologies including wireless metropolitan access networks (WMAN), wireless local access networks (WLAN), and wireless personal access networks (WPAN).
Numeric Communications	Components that perform numeric communications functions such as ADPCM coder, QAM encoder, Viterbi decoder, modulation, demodulation, scrambler, spreader.
Numeric Control	Items that manipulate data flow during simulation: commutators, multiplexers, demultiplexers, upsamplers, and forks.
Numeric Fixed Point DSP	Bit-accurate DSP models (adders, registers, etc.) with behavioral C++ simulation code.
Numeric Logic	Contains Boolean operators, such as and, or, equals, greater than, etc.
Numeric Math	Components that perform math functions, such as adders, multipliers, integrators, log, sine, cosine.
Numeric Matrix	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	Components that perform basic discrete-time DSP functions, such as FIR filter, IIR filter and adaptive filter, and DTFT.
Numeric Sources	Contains sources (items having output only) 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.
Signal Converters	Converts signal (data) types, from one type to another, for example, CxToFloat (complex to floating-point (real)). 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	Data processing components that operate on time-domain baseband waveforms, e.g., multilevel symbol coders and converters, IQ data coders.
Timed Filters	Time-domain lowpass and bandpass analog filters for filtering baseband or RF signals.
Timed Linear	Various linear operations for time-domain analog baseband and RF signals, e.g., waveform delay, split, sum, sample, switch.
Timed Modem	Analog RF modulators, demodulators, and carrier recovery for AM, FM, PM, QAM, QPSK, GMSK, MSK, DQPSK, and Pi/4 DQPSK formats.
Timed Nonlinear	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.
Timed RF Subsystems	RF subsystem components, such as RF combiner, RF modulator, or RF demodulator.
Timed Sources	Time-domain signal generators for baseband and RF signals, e.g., AM, FM, PM, QAM, clock, sinusoid, pulsed, or video.

Integrator Example

This topic is designed for the new user. If you know how to use ADS for Analog/RF Systems design, read quickly through this section noting the differences to ADS Ptolemy, such as the use of sinks, Data Flow controller, and Interactive Controls and Displays components.

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

Opening Example Workspace

1. From the Main window, choose **File** > **Open** > **Example**.
2. Browse to the ADS Examples\Tutorials directory and select integrator_wrk.7zap.
3. Choose a location to unarchive the example.
4. Select Yes to open the workspace.

Open the Schematic

1. Expand the integrator1 tree.
2. Double-click the schematic sign to open the schematic.

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 components:

- From a Palette List You can select items from a palette at the left side of the Schematic window; first select a palette then click on icon(s) in the palette.
- From a Library You can select items by choosing *Insert* > *Component* > *Component Library*. A window opens that displays libraries; select a library then click on component(s) in the library.

Add a Source

1. We will use the Palette List method first. 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 area.
2. Move the crosshairs to the upper left part of the schematic (to the left of the Fork2 component), then click *once.* A symbol representing the source component is placed in the design area. Beneath the symbol is a block of information with the component

name and editable parameters. We will accept the default values.

3. When all components are placed, click the **End Command** arrow on the toolbar, or press **Escape**. The crosshairs disappear.

> **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 area. (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 symbol representing the TkPlot display component is placed in the design area.
4. Click the **End Command** arrow on the toolbar, or press **Escape**. The crosshairs disappear.
5. Close the Component Library dialog box.

Modify Component Parameters

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

- Double-click the component symbol.
- 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.

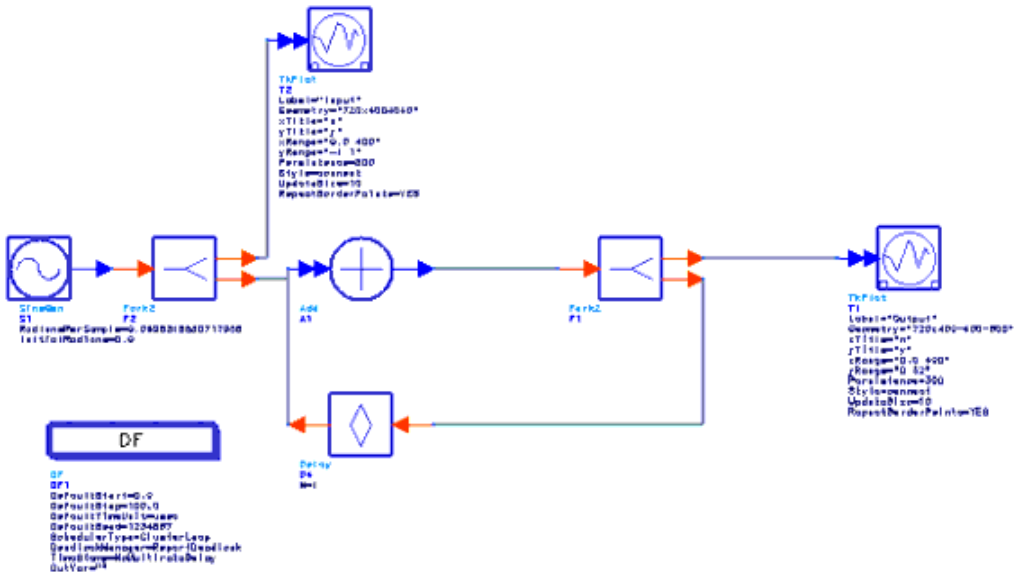
Add a Controller

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 the following figure.



Integrator Design with Source, Display, and Controller Items

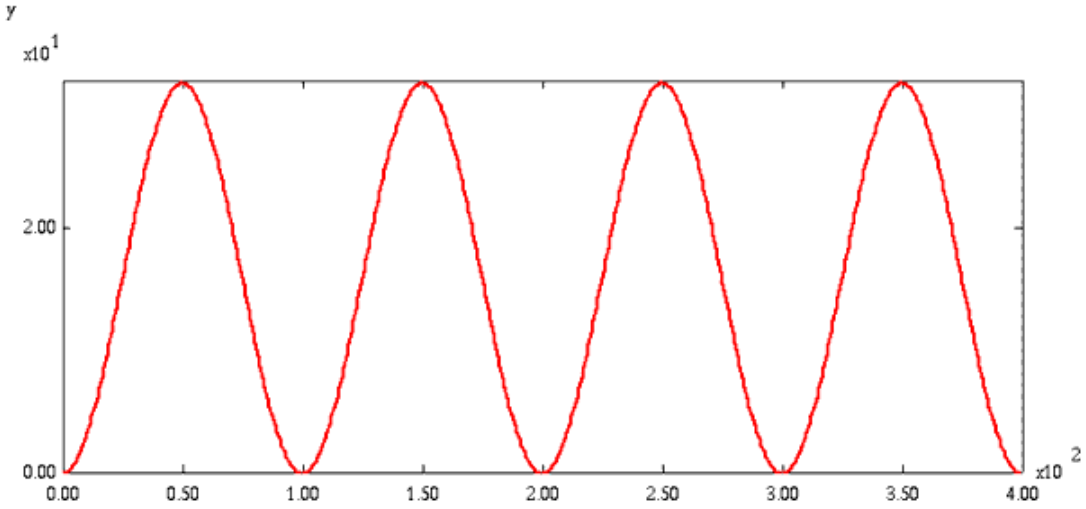
If you have had difficulty building the design, you can select the completed schematic from your directory you copied the example workspace to earlier. Select *integrator1_complete*.

Starting Simulation

Now that we have a completed schematic, we're ready to start a simulation. ADS 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 that provides 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.



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 Ptolemy dialog box 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 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 where you can explicitly name a dataset.
10. In the Dataset field, type **myresults**. This becomes the filename of your simulation results. Accept the other defaults.
11. Choose the **Simulate** button. The simulation begins. A status window appears that gives 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** to dismiss this display.

1. Choose **Window** > **New Data Display**. The Data Display window opens.
2. From the drop-down list, select **myresults**. This list is called the Default Dataset list.
3. Click **Rectangular Plot** in the Plot Types palette list. A ghost rectangular frame appears.
4. Click once to place the frame in the Data Display window. The *Plot Traces & Attributes* dialog box appears:
5. Choose **OK**. Your data is plotted in the Data Display window.

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; Data Display window, which uses stored data and displays it in a variety of formats.

Interactive Controls and Displays for ADS Ptolemy Simulation

The Interactive Controls and Display library components provide real-time simulation input control and animated plots of simulation results. Examples are given below to demonstrate use of these components.
The following table describes two ADS methods for simulating and viewing results.

Simulating and Viewing Results

Method	Description
Interactive Controls and Displays	A quick and easy way to display results. They also give you a way to interactively change parameters while your simulation is running and display animated plots. Data is not saved. No post processing.
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.
Note You can set up a schematic for both methods by placing an Interactive Controls and Displays component and a Sink component.	

The interactive controls and displays were derived from a scripting language called Tool Command Language (Tcl) and the Tool Kit attached to TCL (Tk). *Tcl* or *Tk* are used in these ADS Ptolemy component names. 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. To explore this language further to write your own applications References provides resources regarding Tcl and Tk.

Note
To use Interactive Controls and Displays library components with the ADS tune mode, you must dismiss the Interactive Controls and Displays component between each tune with its pop-up dialog box.

The following table lists the components available in the Interactive Controls and Displays Library.

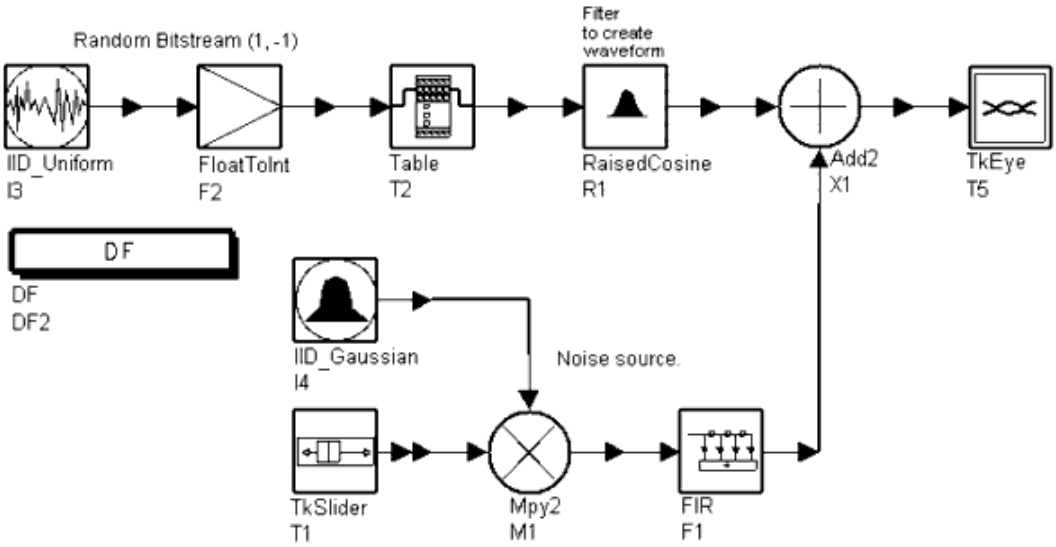
Interactive Controls and Displays Library

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

TkSlider and TkPlot Components

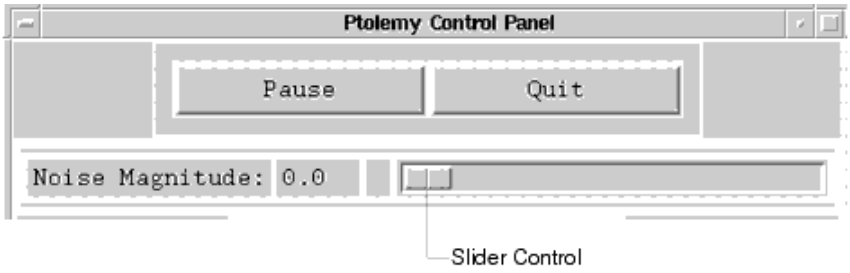
The following figure is taken from the design *File > Open > Example >DSP > dsp_demos_wrk EYE*. To run this example, copy the workspace to a directory for which you have write permission.

The EYE schematic uses an *interactive control* TkSlider component to adjust the amount of noise added to the pulse stream. Simulation results change instantly 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.



Schematic Using TkSlider and TkPlot Components (TkPlot is part of TkEye Subcircuit)

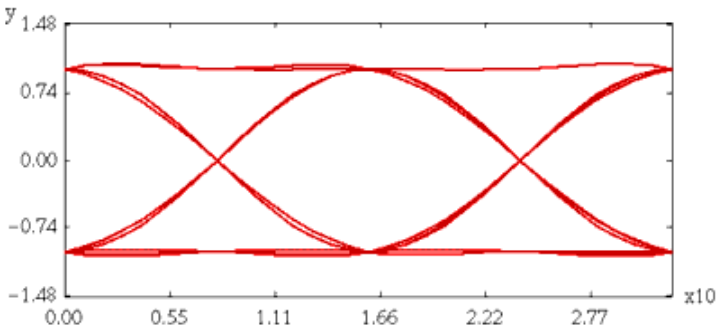
When you begin simulation of a design using TkSlider, an Agilent Ptolemy Control Panel dialog box shown in the following figure is displayed.



TkSlider is Interactively Controlled

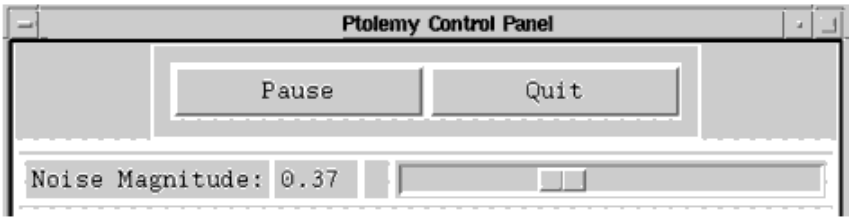
At a minimum, the Pause and Quit buttons are shown. If the TKSlider PutInControlPanel default Yes is accepted, the slider control is available. (If you choose No for this parameter, a separate box will be displayed for each slider.)

With the slider control moved to the far left, 0.0 noise is added; the resulting clean eye-diagram plot is shown in the following figure.

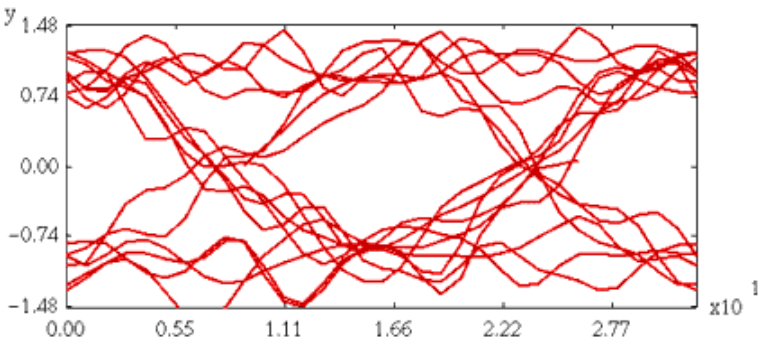


Eye Diagram Plot with Noise at 0.0

If you move the slider so that Noise Magnitude is 0.37 as shown in the following figure, the animated plot changes instantly to display the poor eye diagram shown in the next figure. More or less noise can be added and results will be shown instantly.

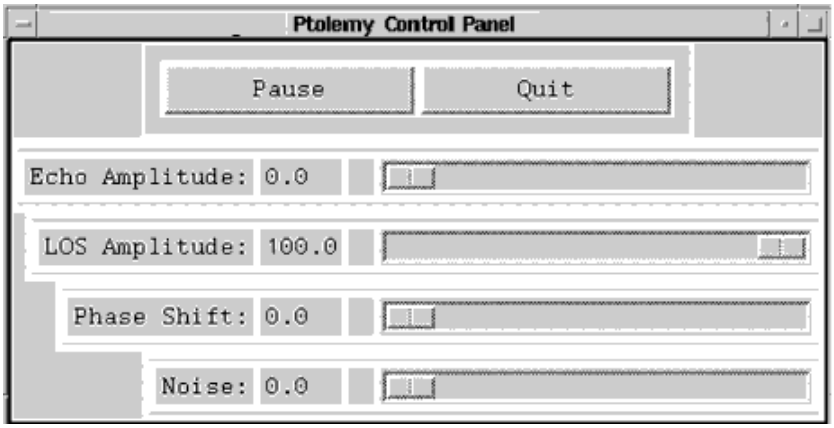


Slider Control Moved to Add Noise



Eye Diagram Plot with Noise at 0.37

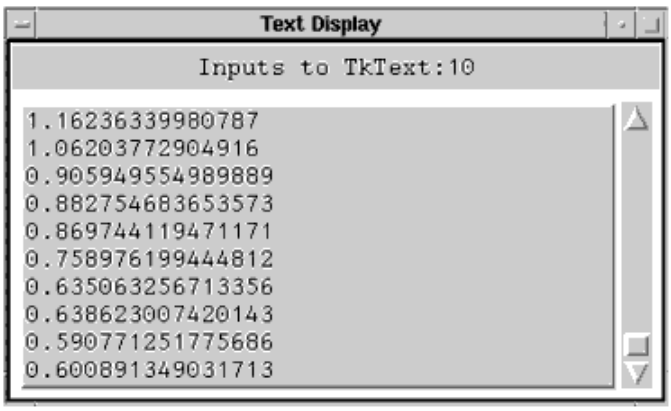
Placing multiple TKSlider components in a schematic will result in multiple sliders in the Agilent Ptolemy Control Panel. You can type a label for each by editing the parameters on-screen or double-clicking the component to bring up the component parameters dialog box.



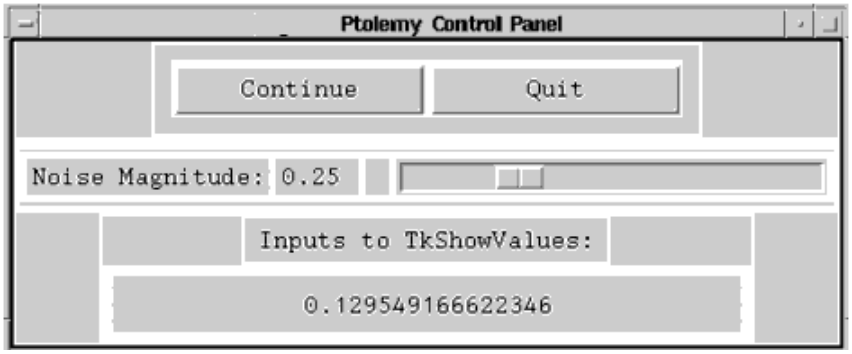
Agilent Ptolemy Control Panel with Multiple TkSliders

TkText and TkShowValues Components

The TkText component simply displays a history of input values in text form.



The TkShowValues component displays only one value is at a time (rather than a history) in the Agilent Ptolemy Control Panel.



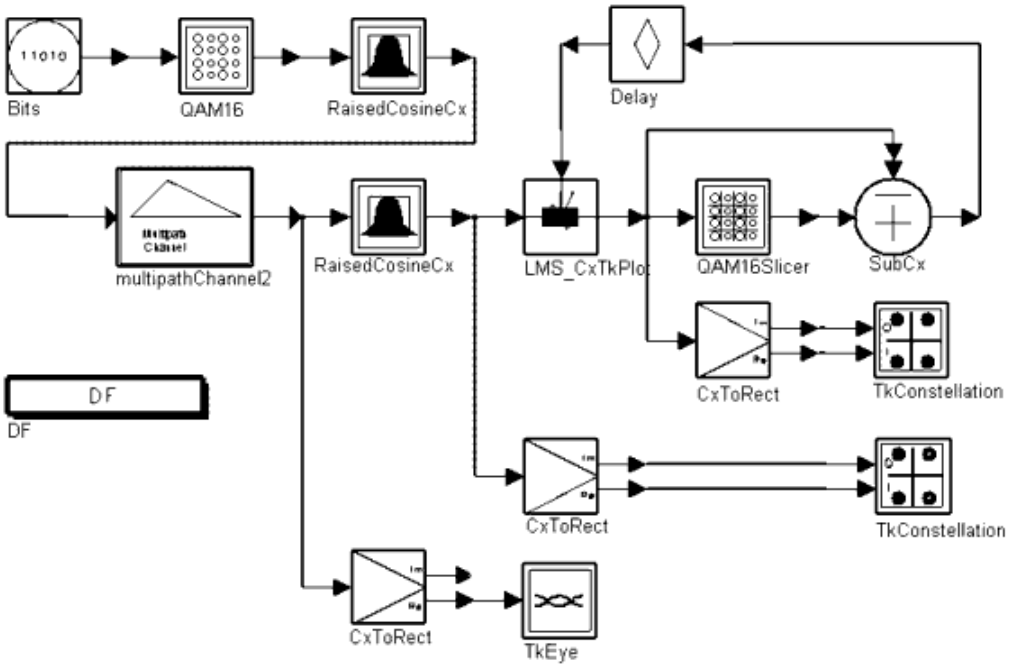
For component and parameter details, refer to *Interactive Controls and Displays* (controls-displays).

TkXYPlot Component

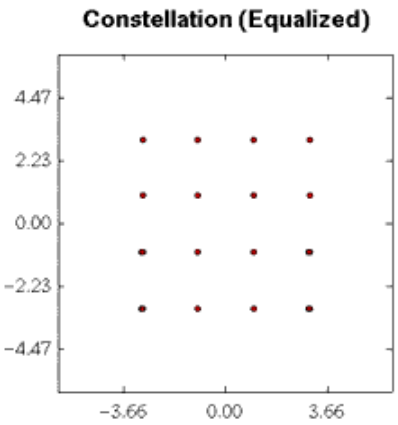
Several TkXYPlot components are placed in this design to show animated results where two inputs, X versus Y, are to be plotted. The schematic shows TkConstellation components which are really subnetworks. Push into the TkConstellation components to see the TkXYPlot components. The TkXYPlot component provides for general purpose plotting of a y input versus an X input with dynamic updating. The TkConstellation component is an application of the TkXYPlot component to achieve a more convenient IQ constellation display.

The example in the following figure is from the *File > Open > Example > DSP > dsp_demos_wrk EQ_16QAM*. To use and simulate this example, copy the workspace to a directory for which you have write permission. In this example TkXYPlot components are used to display constellation diagrams. The equalized constellation diagram is shown in the next figure.

For component and parameter details, refer to *Interactive Controls and Displays* (controls-displays).



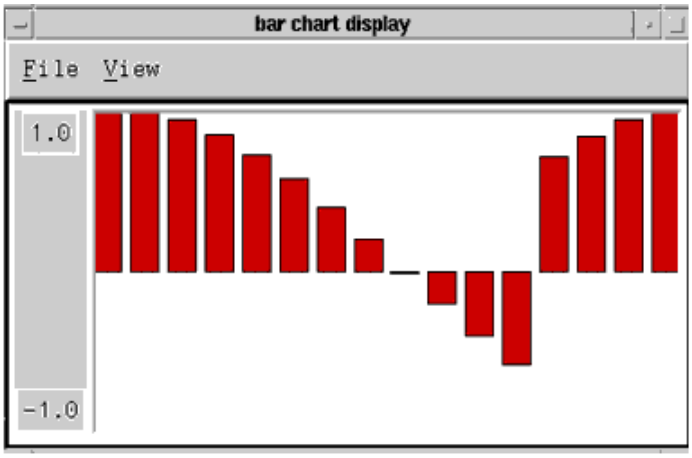
Equalized 16 QAM System with Multipath and Phase Noise



TkXYPlot for Equalized Constellation Diagram

TkBarGraph Component

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



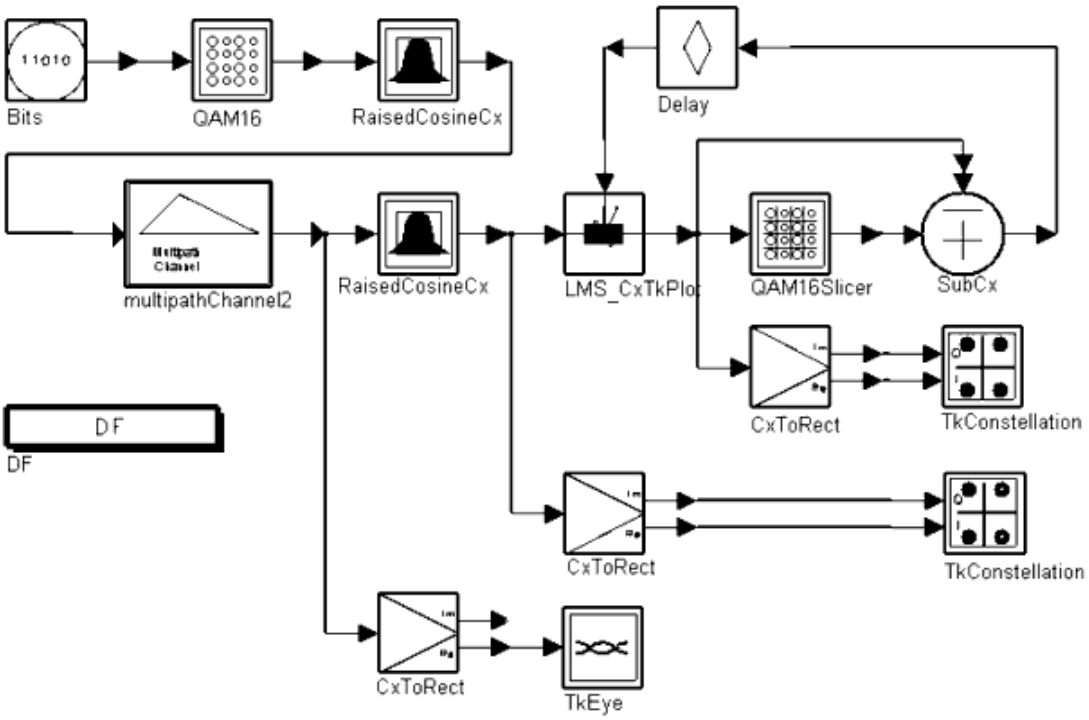
TkBarGraph Display

For component and parameter details, refer to *Interactive Controls and Displays* (controls-displays).

LMS Adaptive Filter Components

There are two LMS adaptive filter components in the Interactive Controls and Displays library: LMS_CxTkPlot and LMS_TkPlot. LMS_CxTkPlot expects complex data and LMS_TkPlot expects real data.

The example shown in the following figure is from the *File* > *Open* > *Example* > *DSP* > *dsp_demos_wrk EQ_16QAM*. To use and simulate this example, copy the workspace to a directory for which you have write permission.



LMS Adaptive Filter (Complex) Component in the 16 QAM System

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

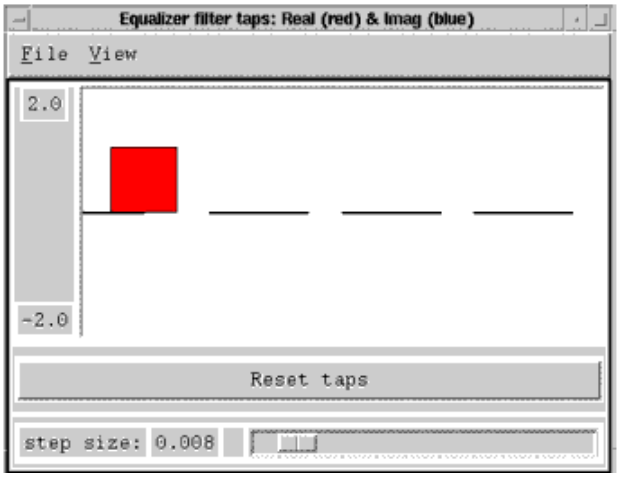
The size of the LMS filter is determined by the number of coefficients in the Taps parameter; the default gives an 8th-order, linear-phase lowpass filter. LMS supports decimation, but not interpolation.

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.

The plot generated from this design upon simulation is shown in the following figure.



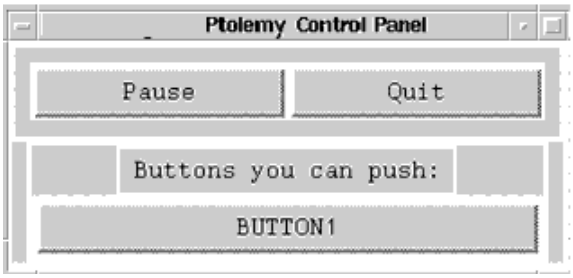
Plot Resulting From LMS_CxTkPlot FilterTaps Parameter

For component and parameter details, refer to *Interactive Controls and Displays* (controls-displays).

TkButtons Component

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 component and parameter details, refer to *Interactive Controls and Displays* (controls-displays).

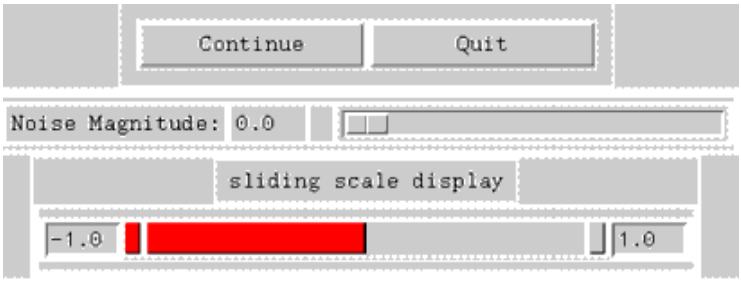
TkBreakPt Component'

With TkBreakPt you can 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.

For component and parameter details, refer to *Interactive Controls and Displays* (controls-displays).

TkMeter Component

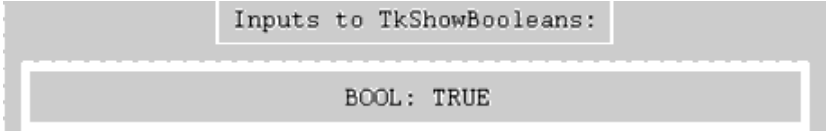
TkMeter dynamically displays the value of any number of input signals (any type) on a set of bar meters. These values are displayed in the Agilent Ptolemy Control Panel.



For component and parameter details, refer to *Interactive Controls and Displays* (controls-displays).

TkShowBooleans Component

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

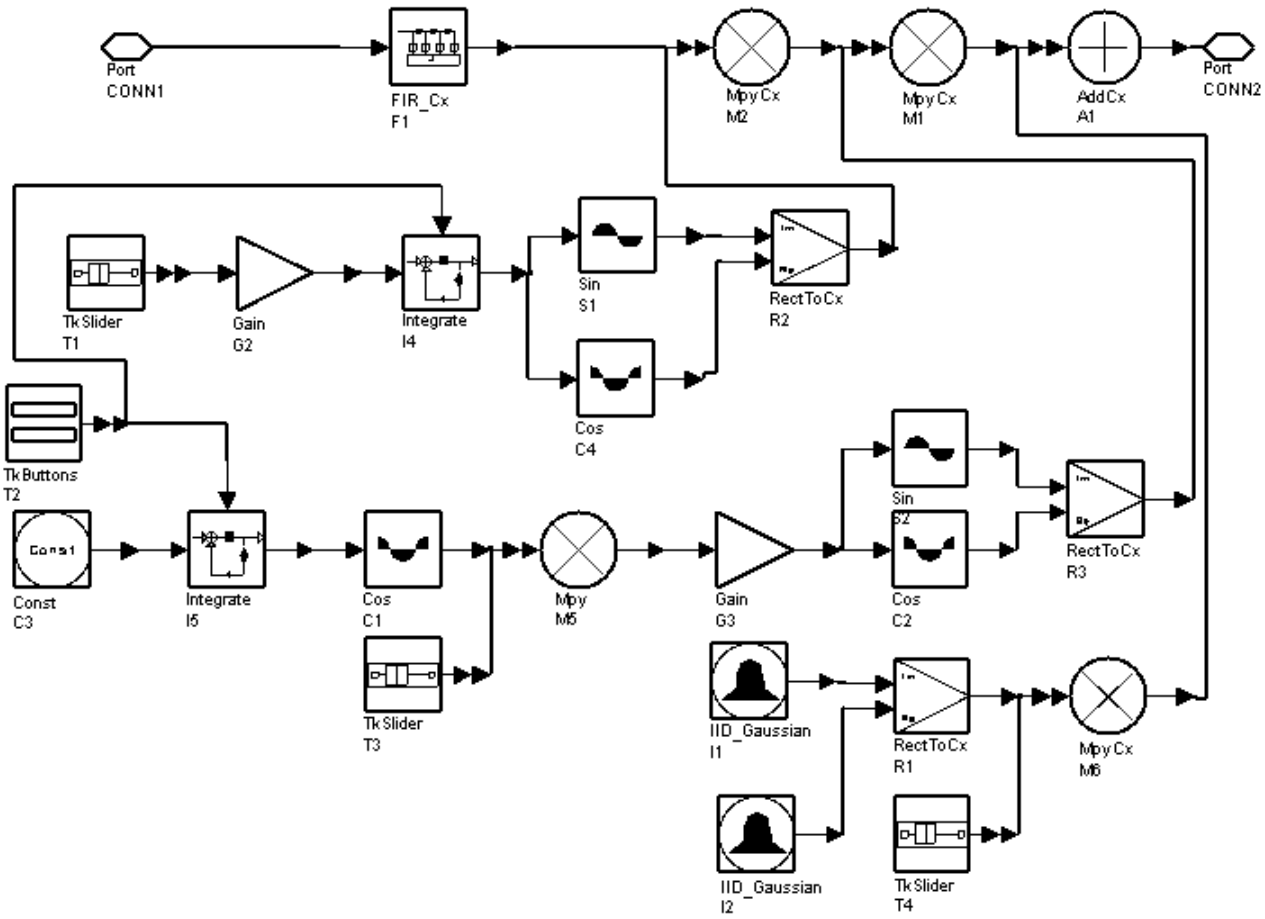


For component and parameter details, refer to *Interactive Controls and Displays* (controls-displays).

TkBasebandEquivChannel Component

The TkBasebandEquivChannel subcircuit, shown in the following figure, models a baseband equivalent channel with linear distortion, frequency offset, phase jitter, and additive white Gaussian noise. Many of the channel model parameters can be set dynamically.

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. 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 frequency offset and 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.



TkBasebandEquivChannel Subnetwork

For component and parameter details, refer to *Interactive Controls and Displays* (controls-

displays).

TclScript Component

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. 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 inputs to the component or set output values. The Tcl script can optionally define a procedure to be called by ADS Ptolemy for every simulation of the component.

For component and parameter details, refer to *Interactive Controls and Displays* (controls-displays).

TkEye, TkConstellation, TkHistogram, TklQrms, and TkPower Components

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 component and parameter details, refer to *Interactive Controls and Displays* (controls-displays).

References

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.

The following web site is a good place to look for Tcl/Tk information:
http://www.yahoo.com/Computers_and_Internet/programming_and_development/languages/tcl_tk/

MATLAB Cosimulation

The MATLAB® models provide an interface between ADS Ptolemy and MATLAB, a numeric computation and visualization environment from The MathWorks, Inc. All ADS Ptolemy *MATLAB Cosimulation models* (matlabblocks) can be found in the *Numeric Matlab* palette and library on the DSP Schematic.

ADS Ptolemy handles the conversion of data to and from MATLAB. Since MATLAB Cosimulation involves multiple environments and associated inter-process communication, the installation and pre-simulation configuration must be precise and correct for the infrastructure to work as expected. To ensure proper operation, the instructions provided in *Setting Up MATLAB Cosimulation* (examples) must be followed.

Supported MATLAB Versions

ADS Ptolemy MATLAB Cosimulation requires MATLAB and works with most of the recent MATLAB versions. Please see *ADS Release Notes* (relnotes) for the list of MATLAB versions confirmed to work with ADS. However, unlisted versions may and most likely will work as well.

- MATLAB Cosimulation is supported in 32- and 64-bit mode on Windows and Linux.
- 32-bit ADS works with 32-bit MATLAB and 64-bit ADS works with 64-bit MATLAB. Mixing 32- and 64-bit modes is not supported.
- MATLAB Cosimulation is supported only through 32-bit compatibility mode on Sun Solaris, i.e. MATLAB version 7.4 (Release 2007a) and newer are not supported on Sun Solaris.

Please see MATLAB Release Notes for information on compatibility between different MATLAB versions.

Setting Up MATLAB Cosimulation

MATLAB must be configured correctly before using cosimulation. If a MATLAB model is run and MATLAB is not configured correctly, ADS Ptolemy will report an error. See *On Windows* (examples) or *On UNIX* (examples) for platform-specific configuration information.

On Windows

For most Windows users, ADS Ptolemy MATLAB Cosimulation will work as expected when MATLAB and ADS are installed.
Typically, the MATLAB installer registers the COM components in the Windows registry.

To manually register COM components run

```
matlab /regserver
```

This may be necessary if you have multiple versions of MATLAB on your system and ADS Ptolemy MATLAB Cosimulation fails with an error *Matlab could not be invoked*.

On UNIX

The configuration process uses the configuration file *hpads.cfg*. For more details on the *hpads.cfg* file, see *Customizing the ADS Environment* (custom).

Set the MATLAB configuration variable to point to the root of your MATLAB installation. An example of this setting is:

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

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. Typically, you will not need to set the MATLABCMD variable.

Alternatively, you can configure all the variables required for MATLAB cosimulation without using the *hpads.cfg* file. In this case, you must configure the following environment variables. The example commands below assume you are using a C-shell or equivalent; modify this command as needed for other shells.

- Set an environment variable named MATLAB that points to the root of your MATLAB installation as shown in this example:

  ```
  setenv MATLAB /usr/local/matlab
  ```
- 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

  ```
  setenv MATLABCMD matlab -c /path/to/license/file
  ```

  so that *matlab* correctly finds its license file. Typically, you will not need to set the MATLABCMD variable.
- Set an environment variable named MLM_LICENSE_FILE to point to the MATLAB license file server as shown in this example:

  ```
  setenv MLM_LICENSE_FILE myport:myserver.mydomain.com
  ```

Simulating with MATLAB

MATLAB distinguishes between complex matrices and floating-point (real) matrices. ADS Ptolemy distinguishes between models that produce data and models that do not. The following table lists six script-interpreting MATLAB models available in ADS Ptolemy; two produce floating-point (real) matrices, two produce complex-valued matrices, and two are sinks that do not produce data. All models can accept any number of inputs provided that the inputs have the same data type, either floating-point (real) or complex.

ADS Ptolemy Models

Model	Description
Matlab_M	Evaluates a MATLAB expression and outputs results as floating-point (real) matrices.
MathlabF_M	Evaluates a MATLAB script file and outputs results as floating-point (real) matrices.
MatlabCx_M	Evaluates a MATLAB expression and outputs results as complex-valued matrices.
MathlabFCx_M	Evaluates a MATLAB script file and outputs results as complex-valued matrices.
MatlabSink	Evaluates a MATLAB expression a fixed number of times.
MatlabSinkF	Evaluates a MATLAB script file a fixed number of times.

These models 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 ADS Ptolemy matrices into and out of MATLAB. Currently, the base model supports 2-D real and complex-valued matrices only.

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

Writing Functions for 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 ADS 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, all "#" characters are replaced with the underscore character "_" because "#" 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 ADS 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 *MatlabSetup* and *MatlabWrapUp* parameters are called during the model's begin and wrap-up procedures. When used in Matlab_M or MatlabF_M components, these parameters can refer to a *. *m* file. During each of these procedures, data is not passed 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.

Hiding MATLAB Code

If you don't want to share your MATLAB IP with other users, you can generate MATLAB p-code files from your m-code by using the MATLAB command *pcode*.

For instance, if your MATLAB code is in the file mycode.m in the data directory of your ADS workspace, you need to open MATLAB and in the MATLAB Command Window use *cd* command to go to that directory and execute

```
pcode mycode.m
```

You will find mycode.p in the same directory. When you run the simulation, mycode.p will be executed instead of mycode.m, and you can remove mycode.m from your ADS workspace, leaving only mycode.p. The format is a non-readable binary, so your m-code is not visible to other users.

Examples

ADS Ptolemy provides example workspaces containing designs demonstrating MATLAB cosimulation. These workspaces are available from the ADS Main window.

For a workspace with designs demonstrating how to call a *MATLAB.m* file, select **File** > **Open** > **Example** > **DSP** > *MATLABlink_wrk*.

- *Channel_Estimate* demonstrates using Matlab_M.

For a workspace with a design demonstrating how to use MATLAB models, select **File** > **Open** > **Example** > **DSP** > *dsp_demos_wrk*.
SOMBRERO is a simple example that plots a SINC function demonstrating how to use

several MATLAB models effectively.

Performing Parameter Sweeps

Performing parameter sweeps with ADS Ptolemy works similar to Analog/RF Network simulators. This section describes this capability using signal processing examples and points out some things to be aware of when performing sweeps with ADS Ptolemy.

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 ADS circuit simulators, sweeps of individual parameters (such as frequency) can be performed from within many of the simulator dialog boxes themselves; in signal processing, a Parameter Sweep (ParamSweep) controller must be used.

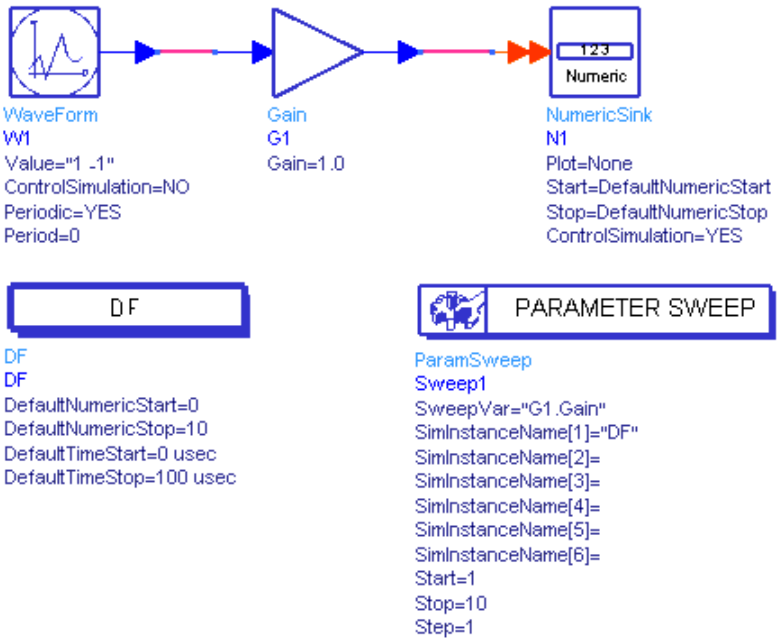
The following sections describe various methods for sweeping parameters.

- For a simple parameter sweep, use a ParamSweep controller. For details, refer to Simple Parameter Sweeps.
- A flexible way to build complicated sweep relationships uses a VAR component to define variables and equations. For details, refer to Parameter Sweeps with Defined Variables.
- 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 SweepPlan in your schematic. For details, refer to Multiple Parameter Sweeps.
- To sweep complex, precision, array, string, or filename 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 is because the simulator only sweeps numbers and these parameter types are *strings* that are interpreted by the simulator. For details, refer to String Type Parameter Sweeps.
- To sweep two or more variables and observe the composite results (for example, analyzing the bit error rate of a communication system for two modulation schemes at three different power levels) uses two ParamSweep controllers. For details, refer to Multidimensional Parameter Sweeps.

Simple Parameter Sweeps

To sweep parameters, place and specify parameters for the ParamSweep controller. Optionally, you can also place a VAR (variables and equations) to aid in defining terms; for details, refer to Parameter Sweeps with Defined Variables.

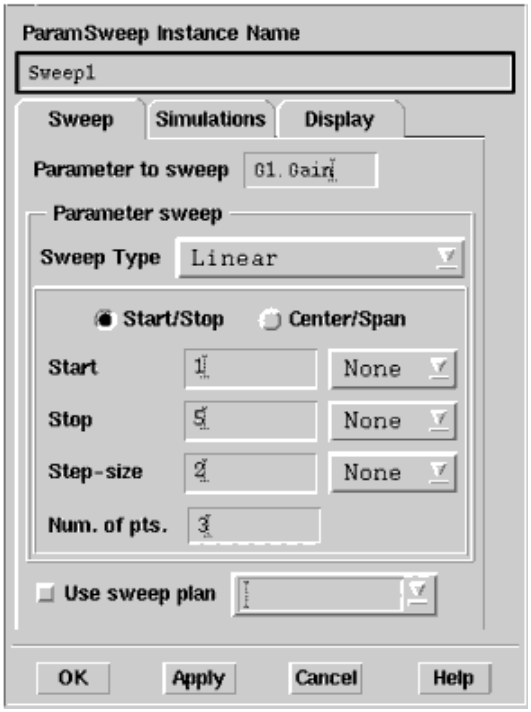
The example design in Sweeping a Gain Component Using ParamSweep includes a Waveform source, a Gain component (Common Components library), a NumericSink (Sinks library), a ParamSweep, and the required DataFlow (DF) controller (Controllers library). We will sweep the Gain parameter of the Gain component.



Sweeping a Gain Component Using ParamSweep

To sweep the Gain parameter of the Gain component:

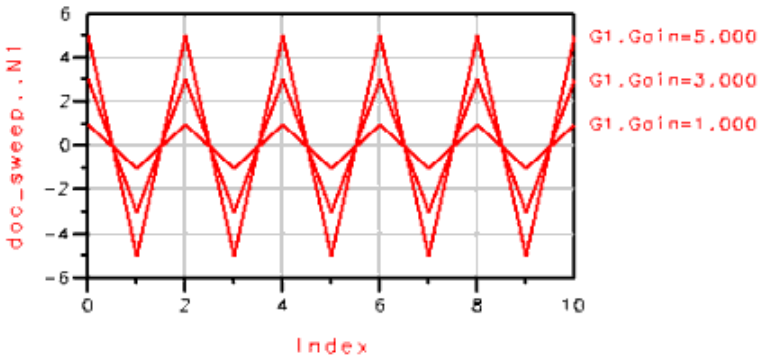
1. Double-click **ParamSweep** symbol to display the dialog box.



2. In the Sweep tab, *Parameter to sweep* field, enter **G1.Gain**. A period separates the instance name (G1) from the parameter we want to sweep (Gain). With this syntax, you can set up any parameter of any component for sweeping. This syntax is hierarchical; if the Gain parameter was in a subnetwork called A, you would use A.G1.Gain.
3. *Sweep Type* field is 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. In the Simulations tab, *Simulations to perform* field, in the Simulation 1 field enter **DF** (this field must be specified for successful simulation). ADS will use the DataFlow controller (instance name DF) as the simulator.
6. Click **OK** to accept your changes and dismiss the dialog box.
7. Double-click the DF symbol to edit its parameters.
8. Change the Stop parameter to **10.0**.
9. Choose **Simulate** > **Simulate**.
10. When the simulation is finished, choose **Window** > **New Data Display**.
11. 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 next.



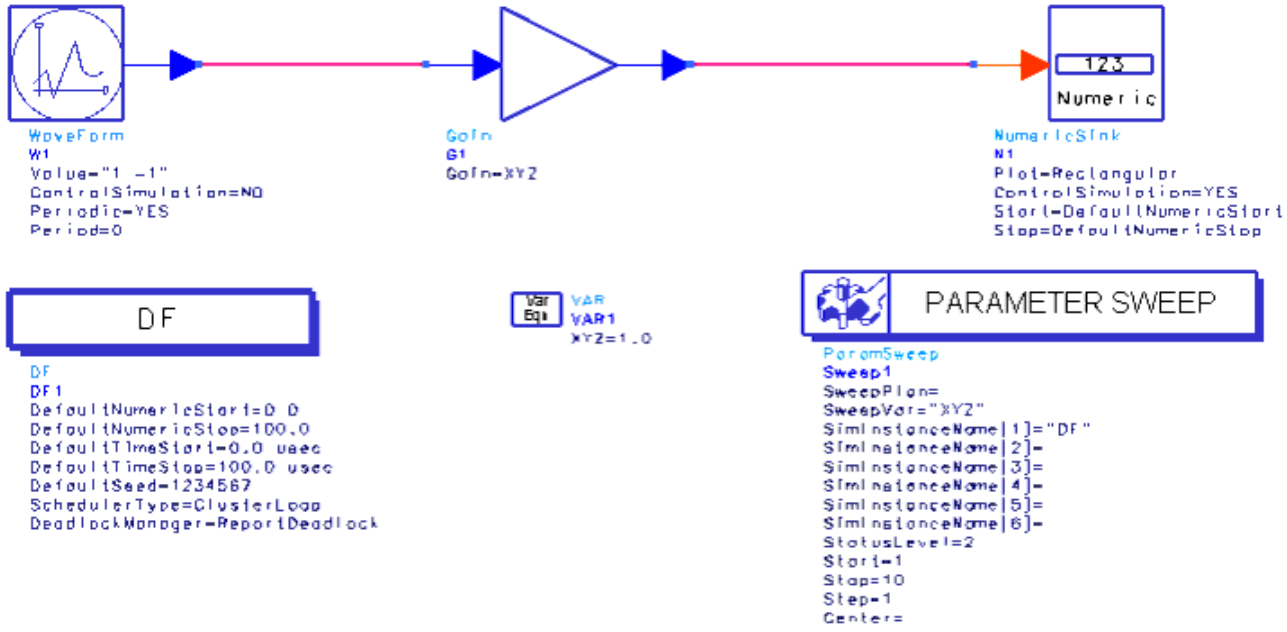
Parameter Sweeps with Defined Variables

An alternate way to conduct parameter sweeps is to place a VAR component in addition to ParamSweep. A VAR component, used to define variables and equations, provides a flexible way of building complicated sweep relationships. (For a simple parameter sweep, it is easiest to use ParamSweep only as described in Simple Parameter Sweeps.)

The following example uses the same design as in the figure above, Sweeping a Gain Component Using ParamSweep, adding a VAR component to help perform the parameter sweep.

To sweep the Gain parameter of the Gain component:

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



Sweeping a Gain Component Using ParamSweep and VAR

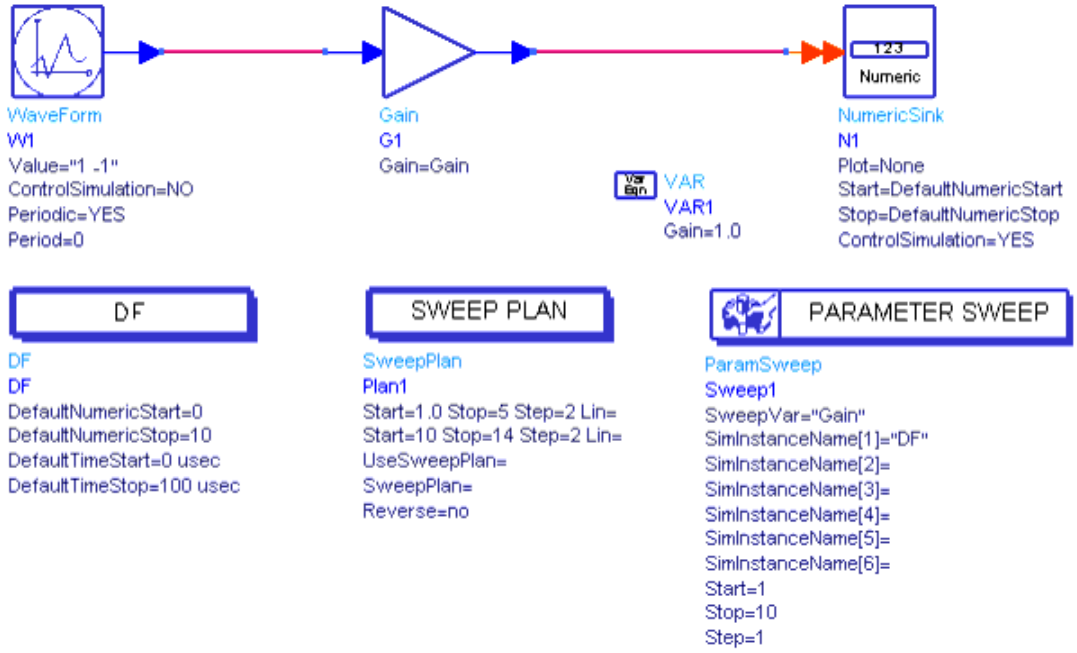
2. Edit the Gain component so that **Gain=XYZ** (instead of the default of Gain=1.0).
3. Edit the VAR component so that **XYZ=1.0**.
4. Edit the ParamSweep controller so that **SweepVar="XYZ"** and **SimInstanceName[1]="DF"**. Do not change other ParamSweep parameters.

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.

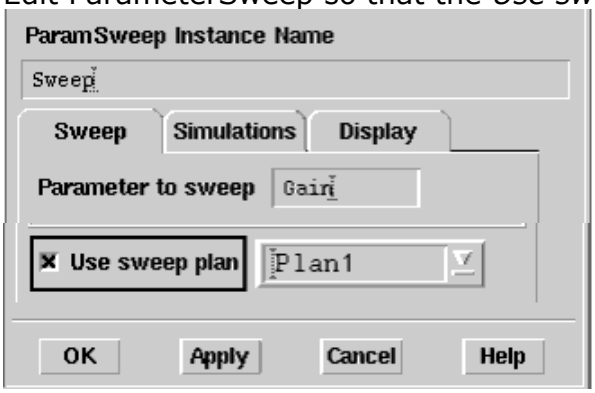
Multiple Parameter Sweeps

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 using a SweepPlan.

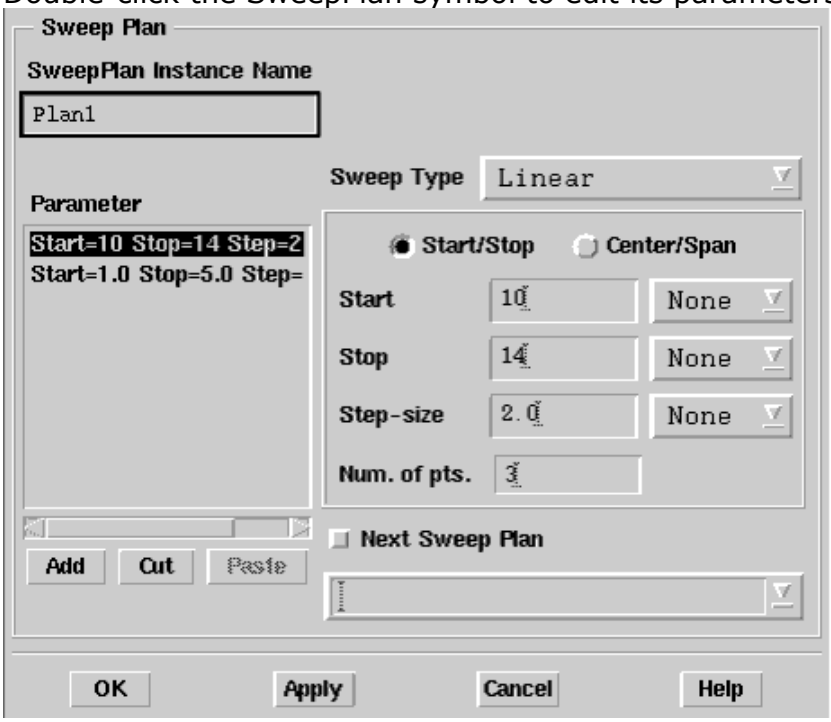
1. Place a SweepPlan (Controllers library) anywhere in the schematic.



2. Edit ParameterSweep so that the *Use sweep plan* check box is selected.



3. Double-click the SweepPlan symbol to edit its parameters.



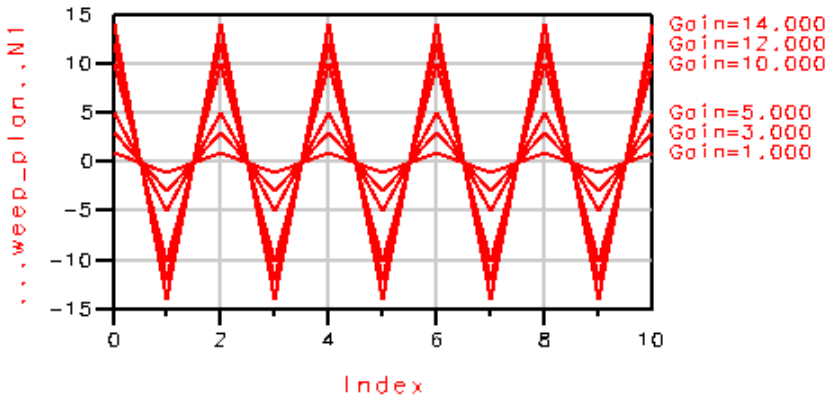
4. Enter two ranges of gain steps. First, in the Start/Stop field repeat the range that was entered for the single parameter sweep:

Start = **1**
Stop = **5**
Step-size = **2**

5. Choose **Apply**.
6. Change the Start/Stop field to:
   Start = **10**
   Stop = **14**
   Step-size = **2**
7. Choose **Add** to add the new range to the sweep plan.
   Click **OK** , or check *Next Sweep Plan* then click **OK** to add more SweepPlan components to control simulation in a chain of events.

> **Note**
> Placement of sweep components 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 SweepPlan. The simulation component calls the first sweep plan to be conducted, whatever it is named.

8. Choose **Simulate** > **Simulate**.
   When the simulation is finished, the Gain parameter has now been swept over two ranges; your Data Display window will look similar to the following:



String Type Parameter Sweeps

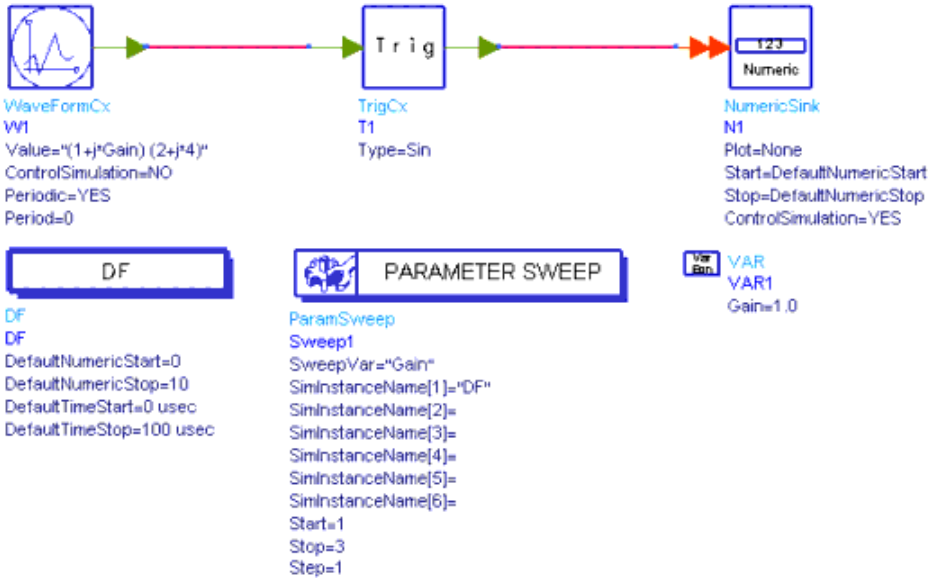
To sweep a real, integer, or fixed-point parameter type, the procedure is similar to examples previously presented. To sweep complex, precision, array, string, or filename parameter types, you must use a VAR (Variables and equations) component to define the swept variable. You then embed a variable from the VAR in the string of the component parameter value. The reason for this is because 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 10 files, *myfile1.dat* through *myfile10.dat* each containing filter coefficients. You might want to sweep a range of these files.

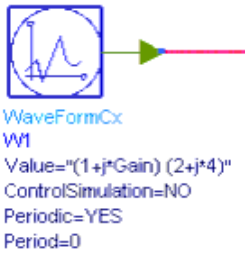
> **Note**
> Earlier versions of Advanced Design System (1.0 and 1.1) required the use of sprintf and strcat functions to reference strings. While no longer needed for complex, precision, or array types, designs built with these functions will still work.

The following example shows how to sweep a *string* parameter type. The example in the following figure uses a VAR (variables and equations) component to define the swept

variable called Gain; the next figure zooms in on the WaveFormCx component, which produces a complex waveform.



Sweeping a Complex Waveform Component Value



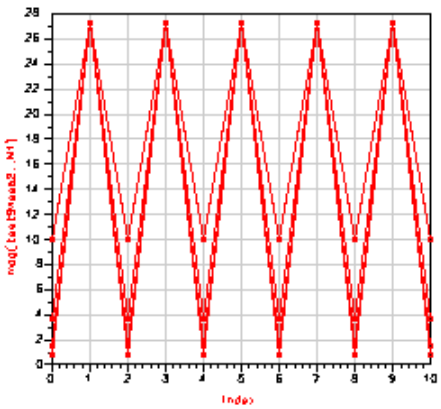
WaveFormCx Component

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 ADS Ptolemy, we sweep the imaginary part as follows:

Value="(1+j*Gain)(2+j*4)"

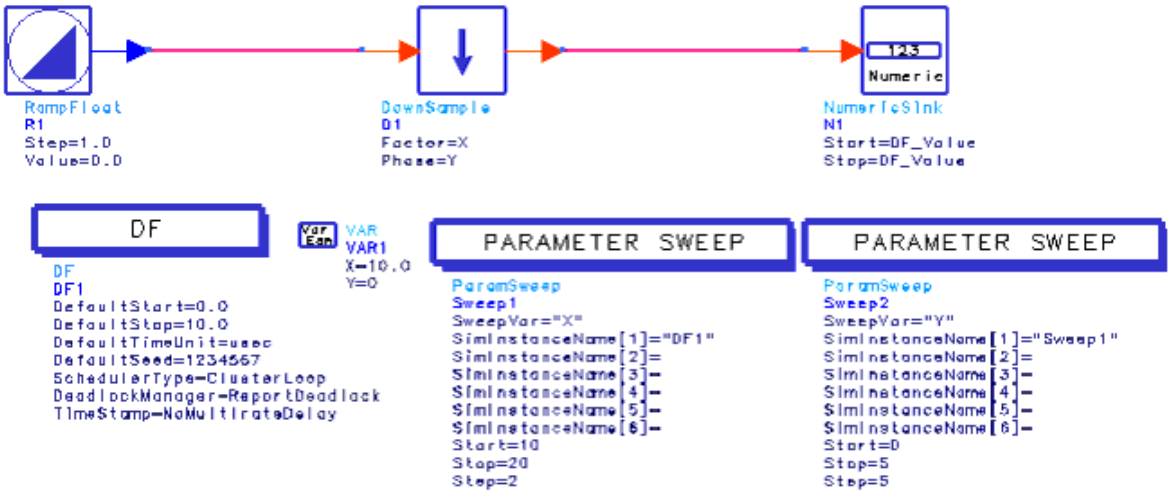
The string "(1+j*Gain)(2+j*4)" tells the software to evaluate the mathematical expression for variable Gain and convert it to a string.

Now if we sweep the variable Gain from 0 to 3, we obtain the following results for the magnitude of the output.



Multidimensional Parameter Sweeps

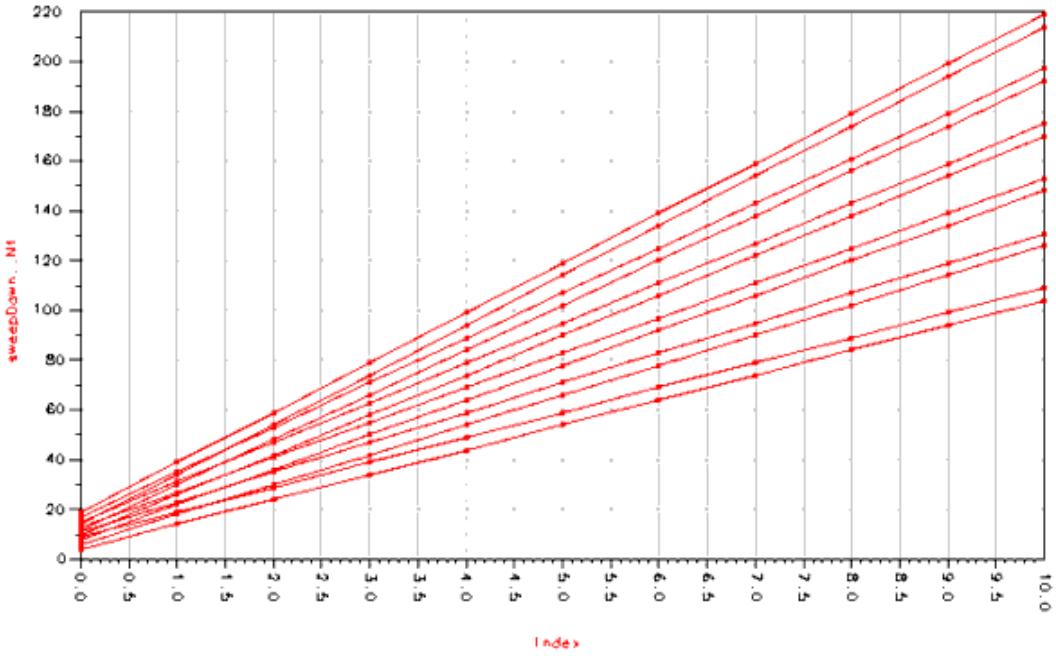
Sometimes you need to sweep two or more variables and observe the composite results. An example is analyzing the bit error rate (BER) of a communication system for two modulation schemes at three different power levels. Here, for each of two modulation formats X, there are three power levels Y. This type of simulation uses two ParameterSweep controllers. An example design is shown in the following figure.



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. DownSample parameters Factor=X and Phase=Y 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.

A total of 6×2 traces are displayed in the Data Display window shown in the following figure. Each line is associated with a downsampling factor (X) for a given phase (Y).



Simulation Results of Multidimensional Sweep Example

SystemC Cosimulation

This documentation describes the ADS SystemC cosimulation feature, including how to import a SystemC model and simulate your design in ADS. Detailed information describes the parameters, theory of operation, and troubleshooting information.

Overview

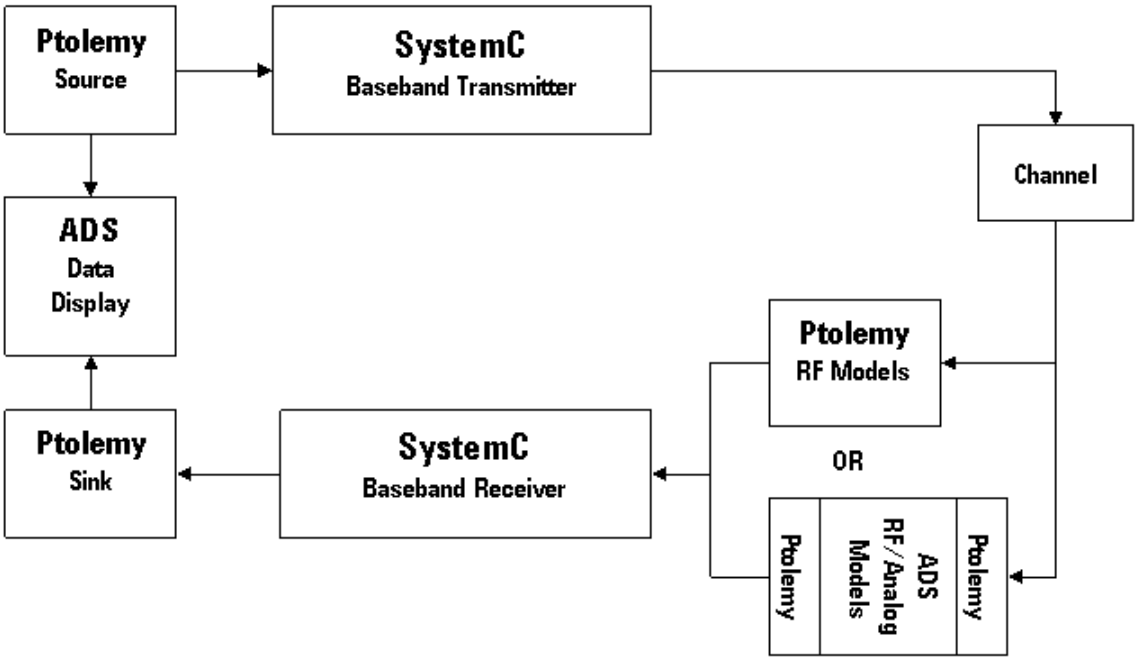
ADS SystemC cosimulation enables you to simulate your Signal Processing SystemC design in the same schematic with other ADS Ptolemy components. This integrated capability provides complete design flexibility complementing other ADS modules and cosimulations, including:

- HDL cosimulation (see *HDL Cosimulation* (hdlcosim))
- MATLAB cosimulation (see *MATLAB Cosimulation Introduction* (ptolemy))
- Cosimulation with Analog-RF Systems (see *Cosimulation with Analog-RF Systems* (ptolemy))

The ability to design all portions of a communications product in one integrated environment can eliminate design errors resulting from disconnects among design teams. By cosimulating with SystemC designs, you can easily incorporate your existing SystemC intellectual property into new designs.

With SystemC cosimulation, you can test baseband designs in SystemC with extensively tested and robust ADS RF Models. Furthermore, SystemC cosimulation with Ptolemy DSP Models, HDL, and MATLAB enables you to validate your design at different level of design abstraction. This cosimulation capability in one design environment makes it easy to test SystemC design along with complex ADS system designs and see the effect on the entire system.

The following figure shows one possible flow of a SystemC cosimulation in ADS. In this flow the baseband design is implemented in SystemC, and the Analog/RF, source, sink, and data display are implemented in ADS.



Supported Platforms

The SystemC cosimulation is available only on following platforms:

- 32-bit Linux, Sun, and Windows versions which are supported by ADS.
- In 32-bit compatibility mode on Linux and Sun versions which are supported by ADS.

SystemC cosimulation is not supported on 64-bit Windows systems.

Supported SystemC Library

ADS includes a pre-compiled OSCI SystemC 2.2 library. SystemC 2.2 implements the SystemC standard as specified in the IEEE 1666 SystemC Standard. This is the only supported form of SystemC library, and it is installed at:

%HPEESOF_DIR%/adsptolemy/lib/<arch>/systemc

where <*arch*> is your computer's operating system (win32, linux_x86, or sun_sparc).

Examples

The *SystemC_Cosim_wrk* contains example designs demonstrating various SystemC cosimulations. The examples are installed with ADS and are located in */examples/SystemC_Cosim/SystemC_Cosim_wrk*. The designs are described in the *Examples Documentation*.

SystemC Cosimulation Capabilities

In ADS, SystemC cosimulation provides the following capabilities:

- **SystemC verification with other ADS Ptolemy models**
  It is often necessary to verify a SystemC design with complete system specification while working at different design abstract levels. The SystemC cosimulation in ADS Ptolemy provides this capability by cosimulating SystemC with other ADS Ptolemy models, such as
  - Native Ptolemy models.
  - RF circuits using a Ptolemy-Analog/RF interface.
  - HDL (Verilog, VHDL) models using HDL cosimulation.
  - MATLAB models using MATLAB cosimulation.

- **Simultaneous cosimulation with multiple SystemC executables**
  It is possible to cosimulate multiple instances of SystemC in ADS Ptolemy. This includes several instances of the same SystemC executable or instances of different SystemC executables.

- **Cosimulation with multirate SystemC models**
  Designs in SystemC can be multirate. In multirate designs, inputs/outputs can consume/produce more than one data point during a single simulation run of a model. The SystemC cosimulation provides support for such SystemC models to be cosimulated in ADS.

- **Passing user-defined parameters from ADS to SystemC**
  A SystemC design often needs to be simulated with different parameter values. With SystemC cosimulation, you can specify parameter values in an ADS schematic and access those values inside SystemC.

- **Custom ADS model interface generation**
  A custom ADS model is generated for every SystemC executable. This includes custom parameters and input/output ports for each SystemC model.

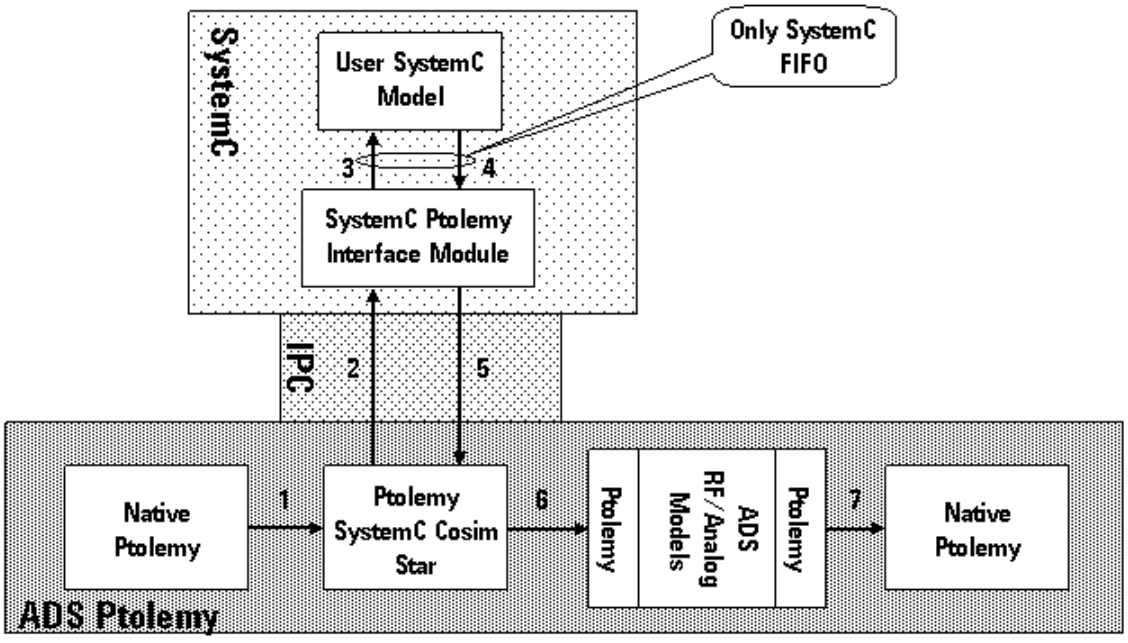
Theory of Operation

The SystemC cosimulation feature uses Inter Process Communication (IPC) for data transfer between ADS Ptolemy and SystemC. This is a two-step use model. In the first step, you import your SystemC model to be cosimulated in ADS Ptolemy, then cosimulate it with other Ptolemy components in the second step.

The SystemC cosimulation requires that one part of the IPC is implemented in ADS and the second part of the IPC is implemented in SystemC. This is achieved by importing your SystemC model into ADS Ptolemy. Importing a SystemC model into ADS requires you to write a simple Ptolemy interface star using the ADS Ptolemy Preprocessor Language in a *SDF<star_name>.pl* file, where < *star_name* > is the name of the custom ADS component for your SystemC model. This *SDF<star_name>.pl* is then used with the ADS Ptolemy model builder to generate a custom ADS component < *star_name* > that can be used in an ADS schematic. This component provides the ADS Ptolemy portion of the IPC for cosimulation. The *SDF<star_name>.pl* is then used to generate a SystemC code in two files named *SystemCPtolemyInterface*<star_name>.hxx_ and *SystemCPtolemyInterface* <star_name>.cxx_, where *SystemCPtolemyInterface*<star_name>_ is the name of SystemC module that provides the SystemC portion of the IPC.

You can use *SystemCPtolemyInterface*<star_name>_ inputs/outputs SystemC FIFOs to access ADS Ptolemy, and its accessor functions to access parameter values that you have specified in the ADS Ptolemy schematic. You will include the *SystemCPtolemyInterface* <star_name>.* files in your SystemC design and compile them with your other SystemC code to generate a SystemC executable. Preferably, you should use the ADS Ptolemy model builder to compile your SystemC code along with *SystemCPtolemyInterface* <star_name>.* files, or use your own *make* system to compile it. In the later case you are required to link against the SystemC library that is shipped with ADS and an ADS Ptolemy specific library. Further details of SystemC import and how to access *SystemCPtolemyInterface*<star_name>_ inputs/outputs FIFOs and user-defined parameters are included in the next section.

Once you have imported your SystemC model into ADS Ptolemy, you can add it to an ADS schematic like any other user-defined Ptolemy component. ADS Ptolemy will execute and cosimulate with the SystemC design automatically. During cosimulation, the data transfer between ADS Ptolemy and SystemC through IPC occurs as shown in the following figure:



In this figure a SystemC model with inputs and outputs is cosimulating with ADS Ptolemy. SystemC source and sink models can also be simulated in the same manner. The following dataflow sequence happens during each simulation run:

1. The Ptolemy SystemC Cosim Star (< *star_name* >) receives input data from other ADS Ptolemy component(s).
2. The Ptolemy SystemC Cosim star sends input data to the SystemC Ptolemy Interface Module (*SystemCPtolemyInterface*<star_name>_) through IPC.
3. The Ptolemy Interface Module then writes data to its outputs interface FIFOs, your SystemC code reads data from these FIFOs and processes it. Improperly reading data from these FIFOs during each simulation run creates deadlocks. Extra care is needed while reading FIFOs in multirate designs.
4. The Ptolemy Interface Module waits for your SystemC code to write data into its input FIFOs. If you do not write data properly to these FIFOs in each simulation run then deadlocks can occur. Extra care is needed in multirate designs to avoid such deadlocks.
5. The Ptolemy SystemC Cosim star receives data from the Ptolemy Interface Module through IPC for further processing in ADS Ptolemy.
6. The Ptolemy SystemC Cosim star sends data to other ADS Ptolemy components.

Steps 1-3 are skipped for SystemC source models with no inputs, and steps 4-6 are skipped for SystemC sink models with no outputs.

Supported Data Types

The ADS SystemC cosimulation currently supports only float and int types for input and output ports, and int, intarray, float, and floatarray for parameter values. However, it is possible to pass string parameters to SystemC using the default *CmdArgs* parameter which passes the string parameters as command-line arguments to SystemC.

The data transfer between the SystemC Ptolemy Interface Module (*SystemCPtolemyInterface*<star_name>_) and your SystemC code is supported by using only double and int type SystemC FIFOs.

Multirate Designs

The ADS SystemC cosimulation supports multirate designs. However, you must be extra careful while accessing *SystemCPtolemyInterface*<star_name>_ inputs/outputs FIFOs. If there are any inputs/outputs that consume/produce multiple data points in a single run, these must be accessed in the same order as you have defined them in your *SDF<star_name>.pl* file. The reason is that *SystemCPtolemyInterface*<star_name>_ writes/reads these input(s)/output(s) in sequential order to its input/output FIFOs, which is the same as in the *SDF<star_name>.pl* file. The SystemC FIFOs perform blocking read/write and if the same order is not followed while reading/writing from/to these FIFOs then an artificial deadlock can be created because of limited FIFO sizes. Another approach is to read/write all the input/output FIFOs in different threads.

Importing a SystemC Model

This section explains how to import a SystemC model into ADS. Before continuing, you must be familiar with the ADS Ptolemy Model Builder, and the ADS Ptolemy Preprocessor Language. For information about these topics, see the following topics in *User-Defined Models* (modbuild):

- *Building Signal Processing Models* (modbuild)
- *Using the ADS Ptolemy Preprocessor Language* (modbuild)

To import a SystemC model into ADS:

1. Write an *SDF<star_name>.pl* file (see Writing an SDF<star_name>.pl).
2. Use the ADS Model Builder to generate a < *star_name* > component for the ADS schematic (see Building a Component for the ADS Schematic).
3. Use the ADS Model Builder to generate *SystemCPtolemyInterface*<star_name>_ SystemC module (see Generating a SystemC Module: SystemCPtolemyInterface_<star_name>).
4. Write *ptolemy_sc_main()* (see Writing ptolemy_sc_main()
5. Use ADS or your *make* system to compile SystemC code including *ptolemy_sc_main()* and *SystemCPtolemyInterface*<star_name>_ (see Compiling SystemC Code).

Writing an SDF<star_name>.pl

Importing a SystemC model into ADS requires that you write an *SDF<star_name>.pl* file using the ADS Ptolemy Preprocessor Language. In this file you will specify the Ptolemy-SystemC interface which includes inputs, outputs, and any user-defined parameters that you would like to specify in the ADS schematic and access in SystemC.

To show an example of a *.pl* file, we'll start with the following SystemC code for an Up Sampler:

MY_SC_UpSample.h

```
#ifndef MY_SC_UpSample_H_INCLUDE
#define MY_SC_UpSample_H_INCLUDE
#include <systemc.h>
SC_MODULE(MY_SC_UpSample) {
 sc_fifo_out<double> output;
 sc_fifo_in<double> input;
 int Factor;
 int Phase;
 double Fill;
 SC_CTOR(MY_SC_UpSample) {
   Factor = 2;
   Phase = 0;
   Fill = 0.0;
   SC_THREAD(go);
 }
 void go();
};
#endif
```

MY_SC_UpSample.cc

```
#include "MY_SC_UpSample.h"
```

```
void MY_SC_UpSample::go() {
 int i, match ;
 match = Factor - Phase - 1;
 while(1) {
   for(i=0; i< Factor; i++) {
     if(i==match)
       output.write(input.read());
     else
       output.write(Fill);
   }
 }
}
```

The SystemC module *MY_SC_UpSample* has one FIFO input, one FIFO output, and three parameters. The corresponding *SDFMY_SC_UpSample.pl* file, where < *star_name* > in this case is *MY_SC_UpSample*, can be written as follows.

SDFMY_SC_UpSample.pl

```
defstar {
 name {MY_SC_UpSample}
 derivedFrom { SystemC_Cosim }
 domain {SDF}
 desc { Data Up Sampler }
 explanation {
   A SystemC Cosim Star. Provides Cosim with a SystemC UpSampler.
   Upsample by a given "factor" (default 2), giving inserted samples
   the value "fill" (default 0.0). The "phase" parameter (default 0)
   tells where to put the sample in an output block. A "phase" of 0
   says to output the input sample first followed by the inserted samples.
   The maximum "phase" is "factor" - 1.
 }
 author { Junaid A. Khan }
 copyright {
 Copyright (c) Agilent Technologies 2007
 All rights reserved.
 }
 vendor { AgilentEEsof }
 location { SystemC }
 input {
   name{input}
   type{float}
 }
 output {
   name{output}
   type{float}
 }
 defstate {
   name {Factor}
   range { [1:inf) }
   type {int}
   default {2}
   desc { number of samples produced }
 }
 defstate {
   name {Phase}
   range { [0:Factor-1] }
   type {int}
   default {0}
   desc { where to put the input in the output block }
 }
```

```
 defstate {
   name {Fill}
   range { (-inf:inf) }
   type {float}
   default {0.0}
   desc { value to fill the output block }
 }
 constructor {
   SystemC_Executable.setInitValue("myscupsample");
 }
 method{
   name {sc_setup}
   access{protected}
   arglist{"(void)"}
   code {
     if (Factor <= 0) {
       Error::abortRun(*this,"Value of Factor must be greater than 0");
     }
     else
       output.setSDFParams(int(Factor),int(Factor)-1);
     if (Factor > 0 && Phase >= Factor)
       Error::abortRun(*this,"Value of Phase must be less than Factor");
     if (Phase < 0)
       Error::abortRun(*this,"Value of Phase must be greater than 0");
   }
 }
}
```

With SystemC import, the purpose of the *SDF<star_name>.pl* file is to define the interface between ADS Ptolemy and SystemC; therefore, only a subset of the ADS Ptolemy Preprocessor Language is used in this file. In this subset, some *defstar* items are required and some are optional. The following table describes the *defstar* items supported for SystemC cosimulation, indicating whether they are required or not.

Defstar Items Supported for SystemC Cosimulation

Defstar Item †	**Description**
name	(required) Name of the star. The defstar example above, SDFMY_SC_UpSample.pl, defines *name* as *MY_SC_UpSample.*
derivedFrom	(required) All stars for non-sink SystemC models must be derived from *SystemC_Cosim* ; stars for SystemC sink models must be derived from *SystemC_CosimSink*.
domain	(required) Domain must be SDF.
desc	(optional) This item defines a short description of the star. This description is displayed by the Advanced Design System design environment for this star in the Library list. It use this syntax: `desc { text }`.
explanation	(optional) This item is used to give a longer explanation of the star's function.
author	(optional) Author's name.
copyright	(optional) Copyright information.
vendor	(optional) Vendor's name.
location	(required) Component library (palette) name where user will find this component in ADS Ptolemy schematic.
input, output	(optional/required) It is required to have at least one

	input or output. In the case of a sink, only outputs are allowed. The input and output ports contain the following sub-items: name (required) - Input or output port name. The same nam e is used to access this port inside SystemC code. type (required) - Currently only float and int types are supported. desc (optional) - Short description of the port.
defstate	(optional) This item defines a user-defined parameter. The defstate contains the following sub-items: name (required) - Name of the parameter, the same name is used to access this parameter inside SystemC code. type (required) - Must be one of the int, float, intarray, or floatarray type. default (optional) - Could be used to specify default value of the parameter. desc (optional) - Short description of the parameter. range (optional) - This is an optional field for each state parameter, specifying the range of a parameter.
constructor	(required) This defstar item should be used only to specify the executable name of the SystemC model used with the cosimulation. The SystemC executable must include the *SystemCPtolemyInterface_* module that is generated from SDF.pl file. The syntax to specify the SystemC executbale is `SystemC_Executable.setInitValue("ExecutableName");` where *ExecutableName* is the name of SystemC executable. There should not be any space used in this name. It is not required to specify the . *exe* extension on Windows.
method	(optional) The only supported method is *sc_setup* . The user should use this method to set up multirate parameters for input and output ports, and to check the parameter ranges. The parameter ranges could be checked using C++ code in this method, whereas the multirate parameters could be set using following technique. The *sc_setup* method in *SDF.pl* files is used to specify multirate parameters for input/output ports. Use the following syntax to specify the multirate parameter: `name.setSDFParams(multiplicity, past)` where: - *name* is the name of the port specified in input or output defstar-items - *multiplicity* is the number of data points (particles) consumed or produced at each run of this star - *past* must be multiplicity-1 to cosimulate with a multirate SystemC design
† Note: Some of the optional *defstar* items are required in the stars which are derived from *SystemC_Cosim* and *SystemC_CosimSink* to provide SystemC cosimulation. Also, any other *defstar* item that is not mentioned in this table is not supported in SystemC cosimulation. Especially, the use of *setup*, *begin*, *go*, and *wrapup* methods are not supported, and will cause SystemC cosimulation to fail with abnormal OS behavior.	

Building a Component for the ADS Schematic

To build a signal processing model that can be placed in the ADS schematic:

1. Set up area to build models.
   Set up all the environment variables needed to build Ptolemy models, and if you have not set up an area to build ADS Ptolemy models earlier then set up that area as well. If you are not familiar with these steps then read *Building Signal Processing Models* (modbuild). In the remainder of this section it is assumed that your model builder area is set to the *adsptolemy* directory.
2. Write *SDF<star_name>.pl* file.
   Create another directory *adsptolemy/src/pl_files*. You can choose any name instead of *pl_files* for this directory. Write your *SDF<star_name>.pl* file in this directory. To

get started, you can create the *SDFMY_SC_UpSample.pl* with the contents shown in the example above, SDFMY_SC_UpSample.pl.

3. Write make-defs.
   Every directory under *adsptolemy/src* must contain a *make-defs* file. For stars derived from *SystemC_Cosim* and *SystemC_CosimSink*, which is true in the case here, the make-defs file in the directory *adsptolemy/src/pl_files* should contain the following:
   make-defs

   ```
   PTSYSTEMC=1

   PL_SRCS=SDFMY_SC_UpSample.pl

   STAR_MK=my_example
   ```

   The rule `PTSYSTEMC=1` in this make-defs file will instruct the make system to compile *SDFMY_SC_UpSample.pl* and link it to the appropriate libraries needed for SystemC cosimulation.
   The `PL_SRCS` is the space-separated list of *SDF<star_name>.pl* files. You can have multiple *SDF<star_name>.pl* files in this directory, one for each SystemC design that you would like to import in ADS.
   The `STAR_MK` is the name of the shared library that you would like to build for these stars.
   Update the make-defs files at *adsptolemy/src/make-defs* to include following line:

   ```
   DIRS += pl_files
   ```

   This will allow the make system to include the directory pl_files in the next step.
4. Build the shared library.
   To build the shared library and install it into your model builder area, run the following command from the *adsptolemy* directory:

   ```
   hpeesofmake
   ```

   Building the shared library may take some time. If you do a listing of the *adsptolemy* directory, you'll see two new directories, *lib.arch* and *obj.<arch>*, where < *arch*> is your system architecture (win32, linux_x86, or sun_sparc).
5. 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 *adsptolemy* directory:

   ```
   hpeesofmake ael

   hpeesofmake bitmap

   hpeesofmake symbol
   ```

Generating a SystemC Module: SystemCPtolemyInterface_<star_name>

The next step in SystemC import is to generate a *SystemCPtolemyInterface* <star_name>_ SystemC module for you to include in your SystemC design to

communicate with ADS Ptolemy. To generate this module, change your directory to *adsptolemy/src/pl_files*, where *pl_files* is the name of the directory where you have created *SDF<star_name>.pl* files. From this directory run the following command

```
hpeesofmake pt2sysc
```

This will generate two files for each *SDF<star_name>.pl* file with the names *SystemCPtolemyInterface*<star_name>.hxx_, and *SystemCPtolemyInterface* <star_name>.cxx_ under the *adsptolemy/src/pl_files* directory. In our example it will create *SystemCPtolemyInterface_MY_SC_UpSample.hxx*, and *SystemCPtolemyInterface_MY_SC_UpSample.cxx*. These two files contain an implementation of *SystemCPtolemyInterface*<star_name>_ SystemC module and *int sc_main(int argc, char* argv[])* function. You will write the following function instead of *int sc_main(int argc, char* argv[])* :

```
void ptolemy_sc_main(int argc, char * argv[],
SystemCPtolemyInterface_<star_name> & p)
```

Now copy the above generated *SystemCPtolemyInterface*<star_name>.*_ files to the location where you have your other SystemC code. In our example we create another directory *adsptolemy/src/systemc_files* and copy these two files into that directory. For each SystemC executable, you can have only one *SystemCPtolemyInterface* <star_name>_ module. It is not possible to include multiple *SystemCPtolemyInterface* <star_name>_ modules in the same SystemC executable.

Note
You must regenerate *SystemCPtolemyInterface*<star_name>.*_ files whenever you modify your *SDF<star_name>.pl* file.

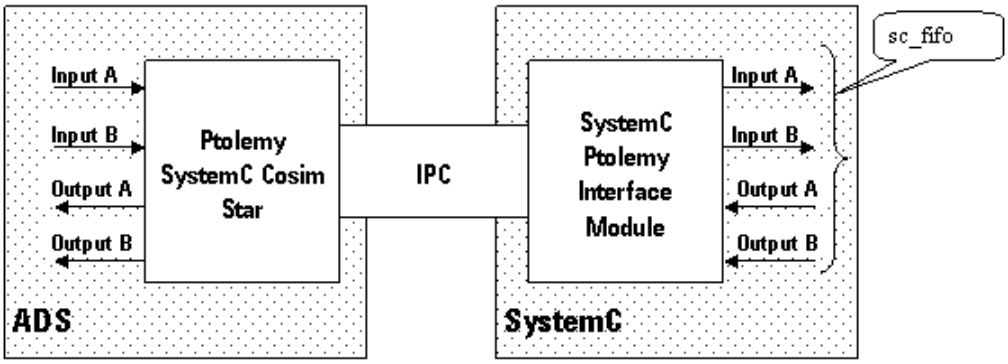
To continue with this *Up Sampler* example, also write the files *MY_SC_UpSample.h*, and *MY_SC_UpSample.cc* into the same directory with the contents shown previously.

Writing ptolemy_sc_main()

The SystemC cosimulation in ADS does not allow you to write *int sc_main(int argc, char* argv[])*, this function is automatically included in the *SystemCPtolemyInterface* <star_name>.cxx_ file. Instead, you are required to write the following function:

```
void ptolemy_sc_main(int argc, char * argv[],
SystemCPtolemyInterface_<star_name> & p)
```

This function is similar to *sc_main* except, it is of type void and has one extra parameter: *SystemCPtolemyInterface*<star_name> & p_. You can use this parameter to access its input/output FIFOs, which will be connected to the corresponding input/output port in your ADS Ptolemy star. The following figure shows the relationship between ADS Ptolemy Star input/output port names and SystemCPtolemyInterface_<star_name> input/output FIFO names.



Note
Inputs to an ADS Ptolemy Star are outputs of SystemCPtolemyInterface_<star_name> and inputs to your SystemC design. Similarly, outputs of an ADS Ptolemy Star are inputs to SystemCPtolemyInterface_<star_name> and outputs of your SystemC design.

Accessing ADS Ptolemy Star Ports inside SystemC

The ADS Ptolemy Star input/output ports can be accessed through input or output FIFOs of *SystemCPtolemyInterface*<star_name> & p_. To access these FIFOs, use *p.portName*, where *portName* is the name of the corresponding port that you have specified in your *SDF<star_name>.pl* file. Please make sure that the type of a FIFO that you are connecting to *p.portName* must match with the type of the port specified in *SDF<star_name>.pl*. You can also read *SystemCPtolemyInterface*<star_name>.hxx_ for details; however, editing this file and the corresponding *.cxx* file is not supported.

Accessing Parameter Values

The *SystemCPtolemyInterface*<star_name>_ module provides two accessor functions to access parameter values you have included in *SDF<star_name>.pl*. You can change the values inside the ADS schematic and read the values using the following accessor functions.

Accessing int or intarray parameter

```
p.getIntegerParamValue(p.paramName); // accessing int parameter
p.getIntegerParamValue(p.paramName, int index); accessing intarray
parameter
```

Accessing float or floatarray parameter

```
p.getDoubleParamValue(p.paramName); // accessing float parameter
p.getDoubleParamValue(p.paramName, int index); // accessing floatarray
parameter
```

where `index` is the index of the array parameter's value starting with index 0.

sc_start()

You must call *sc_start()* inside *ptolemy_sc_main()* to start SystemC simulation. Non-sink models must call *sc_start()* to run the simulation forever; ADS issues a stop command in

such cases and controls the simulation. However, SystemC sink models can stop a simulation any time.

The following SystemC code is written for the *Up Sampler* example:

ptolemy_sc_main.cxx

```
#include <systemc.h>
#include "MY_SC_UpSample.h"
#include "SystemCPtolemyInterface_MY_SC_UpSample.hxx"
void ptolemy_sc_main(int argc, char * argv[],
SystemCPtolemyInterface_MY_SC_UpSample & p){
   sc_fifo<double> sc2pt;
   sc_fifo<double> pt2sc;
   MY_SC_UpSample upSample("UpSample");
   p.output(sc2pt);
   upSample.output(sc2pt);
   p.input(pt2sc);
   upSample.input(pt2sc);
   /// Reading Up-Sampler paramters specified in ADS-Ptolemy Schematic
   /// These includes Factor that specify the multi-rate property of the
   /// Up-Sampler
   upSample.Fill = p.getDoubleParamValue(p.Fill);
   upSample.Factor = p.getIntegerParamValue(p.Factor);
   upSample.Phase = p.getIntegerParamValue(p.Phase);
   sc_start(-1);
}
```

After writing the *ptolemy_sc_main.cxx* file, store it in the directory *adsptolemy/src/systemc_files*.

Compiling SystemC Code

After creating the necessary files, you can compile the SystemC code using the ADS make system or any SystemC-supported make system as described here.

Using the ADS Make System to Compile SystemC Code

ADS provides a *make* system to compile the SystemC code including *SystemCPtolemyInterface_MY_SC_UpSample.** files. You can compile the SystemC code using the ADS Ptolemy model builder area. The previous sections discussed creating the directory *adsptolemy/src/systemc_files* and copying the files listed here into it:

- The following two files contain the SystemC module *SystemCPtolemyInterface_MY_SC_UpSample* that provides the SystemC side IPC functionality:
  - *SystemCPtolemyInterface_MY_SC_UpSample.hxx*
  - *SystemCPtolemyInterface_MY_SC_UpSample.cxx*
- The following file includes the *ptolemy_sc_main()* function:
  - *ptolemy_sc_main.cxx*
- The following two files include SystemC code with which ADS Ptolemy will cosimulate:

- *MY_SC_UpSample.h*
- *MY_SC_UpSample.cc*

To compile these files into a SystemC executable, create the following make-defs file under *adsptolemy/src/systemc_files* :

make-defs

```
SYSTEMC_EXECUTABLE = 1
SRCS = MY_SC_UpSample.cc MY_SC_UpSample.h
SystemCPtolemyInterface_MY_SC_UpSample.cxx
SystemCPtolemyInterface_MY_SC_UpSample.hxx ptolemy_sc_main.cxx
TARGET=myscupsample
```

The rule `SYSTEMC_EXECUTABLE = 1` instructs the make system to compile the files included in the space-separated list SRCS, and link those with the SystemC library and ADS IPC library called *ptsystemclib* to generate an executable defined as *TARGET*.

Be sure that the name of the SystemC executable defined as `TARGET=ExecutableName` is the same as the one that you defined in your *SDF<star_name>* file. This is defined by the constructor method using the following statement:

```
SystemC_Executable.setInitValue("ExecutableName");
```

Also, create another file *make-defs.pre* under *adsptolemy/src/systemc_files* with the following contents

make-defs.pre

```
NO_STL_INC=1
```

The ADS *make* system uses the STL Port by default which conflicts with the SystemC library. Using a *make-defs.pre* file with the line `NO_STL_INC=1` eliminates this conflict. However, using `NO_STL_INC=1` is not supported except while compiling SystemC code unless specified otherwise.

Update the make-defs files at *adsptolemy/src/make-defs* to include the following line:

```
DIRS += systemc_files
```

This allows the *make* system to include the directory *systemc_files* in the next step.

Run the following command from the *adsptolemy* directory

```
hpeesofmake
```

OR

```
hpeesofmake "debug=1"
```

This will generate the SystemC executable at the location *adsptolemy/bin.<arch>*, where *< arch>* is your system architecture (win32, linux_x86, or sun_sparc).

Using Your Make System to Compile SystemC Code

Instead of the ADS *make* system, you can use any SystemC-supported *make* system if you follow these restrictions:

- You are required to use only those compilers supported by ADS.
  In Windows: See *Check the System Requirements* (instalpc).
  In UNIX: See *Check the System Requirements* (install).
- You are required to use the SystemC library shipped with ADS, and installed in these locations:
  In Windows: *%HPEESOF_DIR%\adsptolemy\lib\win32\systemc*
  In Unix: *$HPEESOF_DIR/adsptolemy/lib/<arch>/systemc*
  where < *arch*> is your system architecture (win32, linux_x86, or sun_sparc).
- You are required to link your code with the following ADS-specific library *ptsystemclib* which provides IPC between SystemC and ADS Ptolemy. This library is installed in these locations:
  In Windows: *%HPEESOF_DIR%\adsptolemy\lib.win32*
  In Unix: *$HPEESOF_DIR/adsptolemy/lib.<arch>/*
- On Windows, your code must be built using the Multi-threaded DLL Runtime Library. You can do this by using the */MD* compile flag if you are using a command line *make* system. If you are using Microsoft Visual Studio, you can do this by changing *C/C++/Code Generation/Runtime Library* to *Multi-threaded DLL* (*/MD*) under your workspace properties.

Restrictions on SystemC Code

ADS SystemC cosimulation requires you to follow certain procedures while writing your SystemC code.

Caution
It is important that you follow the coding rules to avoid a possible deadlock between SystemC and ADS Ptolemy and/or a compile error.

Using sc_stop()

You can only use *sc_stop()* if your SystemC design is a sink. A sink has only inputs and no outputs. In this case you must include the following in your *SDF<star_name>.pl* file

```
derivedFrom {SystemC_CosimSink}
```

If you are designing a sink model and you have set the default *ControlSimulation* to *YES* then you must call *sc_stop()* in your design to stop ADS Ptolemy simulation.

Using Non-FIFO Interface

You can use *sc_signal* or any other non- *sc_fifo* interface internally in your design if you follow these rules:

- The interface between your design and the *SystemCPtolemyInterface*<star_name>_

module must use *sc_fifo*.

- ADS Ptolemy is a Synchronous Data Flow (SDF) simulator, which requires the following:
  - During each run of a unirate design, each input reads exactly one data point and each output writes exactly one data point. Your design must follow this convention when reading from or writing to *SystemCPtolemyInterface* <star_name>_ interface FIFOs.
  - Similarly, for multirate designs you must read or write the appropriate number of data points from or to *SystemCPtolemyInterface*<star_name>_ interface FIFOs.

Avoiding Deadlocks

Deadlocks can be avoided by addressing these issues:

- A deadlock can occur by not following the SDF rate conventions while reading/writing data from/to *SystemCPtolemyInterface*<star_name>_.
- Another scenario involves limited SystemC FIFO size. In a multirate design, where the number of data points consumed/produced by a design is more than the FIFO size, a deadlock can occur. To avoid this situation, be sure to read/write FIFOs from/to *SystemCPtolemyInterface*<star_name>_ in the same order in which you have written them in *SDF<star_name>.pl* file. The *SystemCPtolemyInterface* <star_name>_ writes the data to its output FIFOs in the same order in which you have written inputs in *SDF<star_name>.pl*. This means if you do not read these FIFOs in the same order then a blocking write can cause a deadlock. The same can happen while writing the data to *SystemCPtolemyInterface*<star_name>*. Another solution to this problem, which may not always be suitable, is to read/write each FIFO from/to _SystemCPtolemyInterface*<star_name>_ in a separate thread.

Simulating Your Design

Set the following environment variables before starting ADS:

- **ADSPTOLEMY_MODEL_PATH**
  Set *ADSPTOLEMY_MODEL_PATH* 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. Here are examples:
  In Windows: `c:\users\default\adsptolemy`
  In UNIX: `$HOME/adsptolemy`
- **PATH**
  Set *PATH* so it points to the SystemC executable that you have generated during the SystemC import process. Here are examples:
  In Windows: `set PATH = c:\users\default\adsptolemy\bin.win32;%PATH%`
  In UNIX: `setenv PATH $HOME/adsptolemy/bin.<arch>:$PATH`
  using tcsh, where `<arch>` is your system architecture (win32, linux_x86, or sun_sparc).

Now start ADS. You'll see your star appear on the component palette on the left under the location field that you set in your *SDF<star_name>.pl* file. You can use this star as any

other ptolemy star (component). When you simulate your design, this star will start your SystemC executable and cosimulate with it automatically. You can also specify parameter values for this star in the ADS schematic that you are using in your SystemC code.

Default SystemC Cosimulation Component Parameters

A SystemC cosimulation star includes the default parameters shown in the following table, in addition to user-defined parameters.

Default Parameters for SystemC Cosimulation Stars

Parameter	Description
SystemC_Executable	This is the name of the SystemC_Executable you are cosimulating with. This name for each star is tied to the SystemC design for which it was created. The default value for this parameter is set in the SDF<star_name>.pl file.
CmdArgs	This parameter is used to pass command-line arguments to the SystemC design. The command-line arguments can be accessed in this function as *argc* and *argv* : ptolemy_sc_main(int argc, char *argv[], SystemCPtolemyInterface_<star_name>)
SystemC_Timeout	Timeout value in seconds. This parameter is used to detect deadlocks or an abnormal SystemC executable crash.
ControlSimulation	This parameter appears if you are designing a SystemC sink and a corresponding Ptolemy star is derived from SystemC_CosimSink. This parameter tells the ADS Ptolemy simulator whether or not this sink controls the simulation. If you have set ControlSimulation to YES then you must call sc_stop() in your design to stop the ADS Ptolemy simulation. For details, see *Writing Sink Models* (modbuild).

Debugging SystemC Code

This section presents the following steps for debugging your SystemC code. Before continuing here, you should be familiar with the debugging information in *Debugging Your Model* (modbuild).

1. Make sure that your SystemC executable is compiled with the debug flag set to hpeesofmake "debug=1".
2. Set the SystemC_Timeout value to a very large value so that simulation does not time out.
3. Make sure that your simulation will run long enough to perform the following. One way to do this is by using TkPlot component under Interactive Controls and Displays in your design, which asks you to stop the simulation interactively.
4. Start the simulation under the debugger you are using.
5. Attach the SystemC process that you would like to debug.

If you would like the debugger to break inside the SystemC executable before starting a simulation, then use the following method.

1. Write the following in the first lines of *ptolemy_sc_main()* function
```
int i = 0;
while (i==0);
```

2. Recompile the SystemC design using the debug flag set to hpeesofmake "debug=1".
3. Perform the above steps to start simulation in the debugger and to attach the SystemC process.
4. Break the debugger by changing the values of i inside the debugger and continue debugging.

Theory of Operation for ADS Ptolemy Simulation

ADS Ptolemy provides signal processing simulation for ADS's specialized design environments. Each of these design environments capture a model of computation, called a domain, that has been optimized to simulate a subset of the communication signal path. ADS domains that are part of ADS Ptolemy, or can cosimulate with ADS 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 ADS 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 ADS 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. Synchronous dataflow (SDF) and timed synchronous dataflow (TSDF) are described in the sections that follow.

For general documentation on the Circuit Envelope and Transient simulators see *Circuit Envelope Simulation* (cktsimenv) and *Transient and Convolution Simulation* (cktsimtrans). For information on cosimulation with ADS Ptolemy and these circuit simulators, refer to *Cosimulation with Analog-RF Systems* (ptolemy).

Synchronous Dataflow

Synchronous dataflow is a special case of the dataflow model of computation, which was developed by Dennis [1]. 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.

The ADS examples directories contain application examples that rely on SDF semantics. To view these examples, choose **File** > **Open** > **Example**; select the *DSP/dsp_demos_wrk* directory for one group of SDF examples.

SDF is a data-driven, statically scheduled domain in ADS 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.

Basic Dataflow Terminology

The SDF model is equivalent to the *computation graph* model of Karp and Miller [4]. In the terminology of 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.

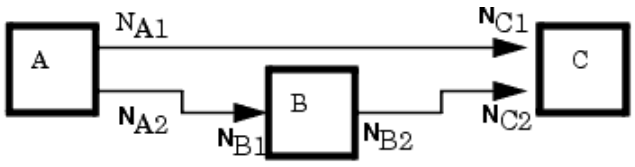
Note
Some ADS Ptolemy terminology differs from UCB Ptolemy terminology. For example, a *component* in ADS is called a *star* in UCB Ptolemy and an *arc* is a *wire*. Refer to the *Terminology* (ptolemy) for more information.

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 ADS Ptolemy. 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.

Consider a simple connection between three components, as shown in the following figure.



Simple Connection of SDF Components Illustrates Balance Equations Constructing a Schedule

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 an 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 the previous figure, 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 Lee [1] that such a solution exists and is unique for every SDF graph that is *consistent* , as defined next.

How Scheduler Works

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.

The multirate scheduler strategically integrates several techniques for graph

decomposition and SDF scheduling. Integrating techniques in this way provides effective, joint minimization of time and memory requirements for simulating large-scale and heavily multi-rate SDF graphs. In practical communication and signal processing systems, single-rate subsystems (subsystems in which all components and interconnections operate at the same average rate) arise commonly, even within designs that are heavily multirate at the global level. The multirate scheduler decomposes a complex multirate graph into a reduced multirate version along with several single-rate subgraphs. These single-rate subgraphs are scheduled in a simple and efficient manner, while the multirate graphs are scheduled using sophisticated graph decomposition and scheduling techniques towards reducing run-time and memory usage.

Iterations in SDF

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

For example, suppose that component B in the figure above, Simple Connection of SDF Components Illustrates Balance Equations Constructing a Schedule, is an FFT_Cx component with its parameters set so that it will consume 128 samples and produce 128 samples. And, suppose that component A produces exactly one sample on each output, and component C consumes one sample from each input. In summary,

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

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

The balance equations become

$$r_A = r_C$$

$$r_A = 128 r_B$$

$$128 r_B = r_C$$

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.

It is not always possible to solve the balance equations. Suppose that in the figure above, Simple Connection of SDF Components Illustrates Balance Equations Constructing a Schedule, we have

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

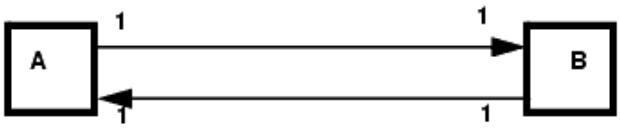
$$N_{B2} = 2$$

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

While scheduling a system, it is possible that all firings specified in the repetition vector will not be completed because none of the remaining components are enabled. Such a system is said to be a *deadlock state*. The following figure illustrates a system in such a state.



Deadlocked SDF System

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 (symbolized by a component with a diamond that is connected to an arc) has a single parameter for 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 ADS 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 ADS Ptolemy, an optional deadlock resolution algorithm can determine where Delay components need to be inserted. If preferred, the delay components can be automatically spliced in.

For the DF (data flow) controller, an Options item DeadlockManager has 3 options:

- Report deadlock The ADS 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, ADS Ptolemy highlights each loop that has deadlocked.
- Resolve deadlock by inserting tokens ADS Ptolemy will automatically resolve the deadlock by inserting delays. This option must be used with care. In general, there are many places ADS Ptolemy can insert a delay to break a deadlock; each can lead to different simulation results.

Timed Synchronous Dataflow

Timed synchronous dataflow is an extension of SDF. TSDF adds a Timed data type (described in *Using Data Types* (ptolemy)). For each token of the Timed type, both a time step and a carrier frequency must be resolved.

ADS examples directories contain numerous application examples that rely on TSDF semantics. To view these examples, choose **File** > **Open** > **Example**, a dialog box appears. Select the *DSP/ModemTimed_wrk* directory for one group of TSDF examples.

Note that the ADS Ptolemy (Pt) simulation time domain signals are different from those used for Circuit Envelope (CE) and Transient (T) simulation.

- For Pt, the simulation time step is not global and the input signals for different components may have a different time step. However, the simulation time step at each node of a design, once set at time=0, remains constant for the duration of the simulation. The time step is initially set by the time domain signal sources or numeric to timed converters. The time step in the data flow graph may be further changed due to upsample (decreases the time step by the upsampling factor) and/or down sample (increases the time step by the downsample factor) components. The time step associated with signals at the inputs of the timed sinks may or may not be the same as the one that was initiated by the timed sources or numeric to timed converters. This is dependent on any up or down sampling that may have occurred in the signal flow graph.
- For CE, the simulation time step is set by the simulation controller and is constant for the duration of the simulation. This is global for all components simulated in the design.
- For T, the simulation maximum time step is set by the simulation controller, but the actual simulation time step may vary for the duration of the simulation. This simulation time step is global for all components simulated in the design.
- For Pt, each timed sink that sends data to Data Display has time values that are specific to the individual sink and may or may not be the same as the time values associated with data from other timed sinks.
- For CE and T, all time domain data sent to Data Display has the same global time value.
- For CE, a time domain signal may have more than one carrier frequency associated with it concurrently. The carrier frequencies in the simulation are harmonically related to the frequencies defined in the CE controller and their translated values that result from nonlinear devices.
- For Pt, a time domain signal has only one characterization frequency associated with it. This characterization frequency is also typically the signal carrier frequency.

However, the term *carrier frequency* is more typically used to mean the RF frequency at which signal information content is centered. A signal has one characterization frequency, but the signal represented may have information content centered at one or more carrier frequencies. This would occur when several RF bandpass signals at different carrier frequencies are combined to form one total composite RF signal containing the full information of the multiple carriers.

Example

Two RF signals and their summation:

```
Arf = Ai*cos(wa*t) - Aq*sin(wa*t) with carrier frequency wa
Brf = Bi*cos(wb*t) - Bq*sin(wb*t) with carrier frequency wb
The summation of these two signal, Crf, can be represented
at one characterization frequency, wc, as follows:
Crf = Ci*cos(wc*t) - Cq*sin(wc*t) with carrier frequency wc
where
wc = max(wa, wb)
Ci = Ai + Bi*cos((wa-wb)*t) + Bq*sin((wa-wb)*t) (assuming wa > wb)
Cq = Aq - Bi*sin((wa-wb)*t) + Bq*cos((wa-wb)*t) (assuming wa > wb)
```

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/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 = UNDEFINED). All Timed ports have an associated $F_c \geq 0.0$. Non-timed ports have an UNDEFINED F_c.

During simulation, all F_c values associated with all 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 ADS 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.

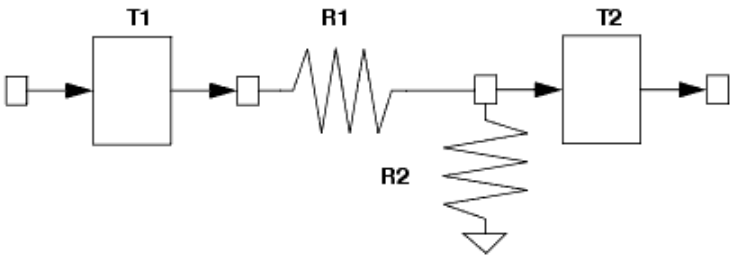
For feedback paths, a default F_C is assigned by the simulator. This default F_C is then propagated until the F_C converges on the feedback path. This F_C is occasionally non-unique. To specify a unique value, use the SetFc Timed component.

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.

The following 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.



TSDF Components with Interconnected Output (R1) and Input (R2) Resistors

Resistors contribute additive thermal noise (kTB) to signals when the specified resistance temperature (RTemp) is greater than absolute zero (-273.15°C) 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.

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-point (real) matrix, fixed matrix, complex matrix, and integer matrix can be used as output from the Printer component.

The following file format examples (real array data through complex matrix data), are drawn from the code and include the *comment line* # symbol.

Real Array Data

```
#  Template for ADS Ptolemy real data
#  Each number separated by new lines
1
0
0
```

Complex Array Data

```
#  Template for ADS Ptolemy complex data
#  Each complex value, (real, imag), separated by new lines
(1.0, 0.0)
(0.0, 0.0)
(0.0, 0.0)
```

String Array Data

```
#  Template for ADS Ptolemy string data
#  Each string value enclosed with double-quote marks "" and separated by new lines
"text 1"
"text 2"
"text 3"
```

Real Matrix Data

```
#  Template for ADS Ptolemy real matrix data
#  Each matrix data set separately listed with brackets around each row and matrix
#  Each matrix row separated by new lines
[[ 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 ADS Ptolemy fixed-point matrix data
#  Each matrix data set separately listed with brackets around each row and matrix
#  Each matrix row separated by new lines
[[ 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 ADS Ptolemy integer matrix data
# Each matrix data set separately listed with brackets around each row and matrix
# Each matrix row separated by new lines
[[ 1  -2,  2 ]
[ -2, 2, -2 ]]
[[ 2, -2,  3 ]
[ -3, 2, -1 ]]
[[ 2, -2,  3 ]
[ -2, 2, -2 ]]
```

Complex Matrix Data

```
#  Template for ADS Ptolemy complex matrix data
#  Each matrix data set separately listed with brackets around each row and matrix
#  Each matrix row separated by new lines
[[ 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 TimedDataRead component. They can be written by the simulator by specifying them as output files in a TimedDataWrite component. The binary format *.sig* file has the same ASCII header information as the *.ascsig* file but data is stored as a pointer in binary format.

- The SPW version 3.0 data file format must be used.
- Comments can only be included on the one line following the $USER_COMMENT statement.
- The TimedDataRead 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
1.00000000000000000000
1.00000000000000000000
-1.00000000000000000000
-1.00000000000000000000
1.00000000000000000000
1.00000000000000000000
END
```

Complex 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 = 10
Signal Type = Double
Complex Format = Real_Imag
$DATA
1.00000000000000000000+j1.00000000000000000000
1.00000000000000000000+j1.00000000000000000000
-1.00000000000000000000+j1.00000000000000000000
-1.00000000000000000000+j1.00000000000000000000
1.00000000000000000000+j1.00000000000000000000
1.00000000000000000000+j1.00000000000000000000
-1.00000000000000000000+j1.00000000000000000000
-1.00000000000000000000+j1.00000000000000000000
-1.00000000000000000000+j1.00000000000000000000
-1.00000000000000000000+j1.00000000000000000000
END
```

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

The general . *tim* file format is:

```
BEGIN  TIMEDATA
#        T        ( SEC   V   R  xx )
```

```
%        t        voltage
<data line>
.
.
.
<data line>
END
```

BINTIM Format

The BINTIM format (. *bintim*) is for binary time-domain waveform data files. In *.bintim* files, the format is the same as *.tim* files, except the BEGIN line is preceded by a line indicating the number of data points, n:

```
NUMBER OF DATA n
```

The <data line> in a *.bintim* file is just a binary dump of all the waveform (time, voltage) data. Also, there is no END line.

Note
The *.bintim* format is not supported in the Data File Tool. However, certain signal processing components can read *.bintim* files.

Guidelines for .tim files

An exclamation point ! at the beginning of a line signifies a comment line; characters that follow ! are ignored by the program.

TIMEDATA data block is required.

When the file reader reads a file, it renames the independent and dependent variable names regardless of the names specified in the file. The file reader reads the independent variable name as *time*, and the dependent variable name as *voltage*.

- The BEGIN statement:

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

- Option line:

  ```
  # T ( time_unit data_unit R xx )
  ```

  where

  # = delimiter tells the program you are specifying these parameters.
  T = time
  time_unit = sets time units. Options are SEC, MSEC, USEC, NSEC, PSEC.
  data_unit = Set the units for the voltage values. Options are:
  V = volts
  MV = millivolts
  R xx = sets resistance, where xx = reference resistance. (default is 50.0)

- Format line:

```
% time voltage
```

where

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

In ADS, the syntax *time* and *voltage* in the Format line are arbitrary. These values can be whatever you prefer. For example, an option line such as:

```
% t mV
```

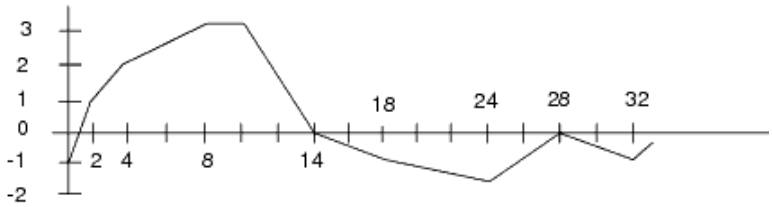
can be used. However, these values are converted to *time* and *voltage* by the file reader when the *.tim* file is imported, and these will be the variables appearing in a dataset (*.ds*) file.

- TIMEDATA data requirements are:
  - A value for time=0 is not required.
  - The signal is assumed to be time periodic with time period equal to maximum time minus minimum time.

Example .tim Files

```
BEGIN  TIMEDATA
# T ( USEC  V  R 50 )
%       time       voltage
        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
```

This example file results in a time periodic voltage versus time with time period 32 µsec, interpreted as a piece-wise linear voltage description.



The following example shows how to handle independent and dependent variable names when using a DataAccessComponent. This is useful since the file reader reads the independent variable name as *time*, and the dependent variable name as *voltage*, regardless of the names specified in the file. The following example data files shows the variable names specified as `t` and `v`:

```
BEGIN TIMEDATA
%      t             v
       0             0
```

```
 1e-011    0.00995017
 2e-011     0.0198013
 5e-011     0.0487706
1.4e-010      0.130642
4.1e-010       0.33635
 1e-009       0.632121
END
```

Though the variable names are `t` and `v`, the file reader changes the names to *time* and *voltage*, requiring the following syntax for the DataAccessComponent:

```
DataAccessComponent
Type=Time Domain Waveform   (TIM MDIF)
iVar1="time"
iVal1=time
VAR
X=file{DAC1,"voltage"}
```

Agilent Standard Data Format (.dat) Files

The *.dat* file is a signal file form used with the 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.

Understanding Parameters

Value Types

ADS Ptolemy requires specific parameter *value types* (string, real array, or complex) for the component parameter values you enter in schematic designs.

Component parameter values can be entered several ways:

- Editing the component parameters dialog box. Double-click the component symbol on the schematic, the dialog box appears. Parameter values can 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.

The following table describes each value type.

Ptolmey 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.
Fixed Point	Parameter editing in Component dialog box: A. Enter real value, but the value used will be based on the precision used with this parameter. 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.
Complex	Editing in Component Parameter dialog box: A. Enter a complex number using the form Re + j * Im. B. Enter expression for a complex value—Example: cos(X)+j*sin(Y), where X and Y are defined expressions, j is the imaginary operator. Parameter editing on schematic: Highlight parameter value on schematic and enter complex value or expression.
String	Editing in Component Parameter dialog box: A. Enter string. Do not enclose this string with any double quote symbols. Note for embedded double quotes ("), use double double quotes (""). B. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for a string value. Parameter editing on schematic: Highlight parameter value on schematic and enter string value enclosed with double quote symbols.

Precision	Editing in Component Parameter dialog box: A. Enter string in the form X.Y or Y/W. Do not 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. B. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for a precision value. Parameter editing on schematic: Highlight parameter value on schematic and enter string value enclosed with double quote symbols.
Filename	Editing in Component Parameter dialog box: 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 ~/, $HOME, $HPEESOF_DIR, or others. B. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for a filename value. Parameter editing on schematic: Highlight parameter value on schematic and enter string value enclosed with double quote symbols.
Integer Array	Editing in Component Parameter dialog box: A. Enter integer values directly—Example: 1 -2 5 2 (spaces separate data). B. Enter values from a file—Example: <filename. If the filename has no path specified, the workspace data directory is used. The content of the file must be numbers separated by spaces or on a new line. For example: 1 -2 5 2 and 1 -2 5 2 are equivalent. C. Enter values directly in addition to file data—Example: 1 <file1 2. If file1 contains -2 5, then the array would be the same as in A. D. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for an integer array. E. Enter values separated by commas, and surrounded by curly braces. Example: @{1,-2,5,2}. Parameter editing on schematic: Highlight parameter value on schematic, then: ▪ enter array values enclosed with double quote symbols or ▪ enter array values as shown in E without the @ and double quote symbols.
Fixed Point Array or Real Array	Editing in Component Parameter dialog box: A. Enter fixed-point values directly—Example: 1.2 -2.3 5.6 2.8 (spaces separate data). B. Enter values from a file—Example: <filename. If the filename has no path specified, the workspace 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. 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. 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. E. Enter values separated by commas, and surrounded by curly braces. Example: @{1.2,-2.3,5.6,2.8}. Parameter editing on schematic: Highlight parameter value on schematic, then: ▪ enter array values enclosed with double quote symbols

	or ▪ enter array values as shown in E without the @ and double quote symbols.
Complex Array	Editing in Component Parameter dialog box: A. Enter complex values directly as ordered pairs separated by a comma (optional spaces may follow the comma). Each ordered pair must be enclosed in parentheses, and separated from other ordered pairs by spaces. Example: (1.2,2.5) (-2.3,1.3) (5.6, -1.4) (2.8, 3.4) B. Enter values from a file—Example: <filename. If the filename has no path specified, the workspace data directory is used. The content of the file must be ordered pairs of numbers separated by a comma (optional spaces may follow the comma). Each ordered pair must be enclosed in parentheses, and separated from other ordered pairs 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. 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. D. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for a complex array. E. Enter values separated by commas, and surrounded by curly braces. Complex values must be entered using the format a+j*b. Example: @{1.2+j*2.5, -2.3+j*1.3, 5.6-j*1.4, 2.8+j*3.4} Parameter editing on schematic: Highlight parameter value on schematic, then: ▪ enter array values enclosed with double quote symbols or ▪ enter array values as shown in E without the @ and double quote symbols.
String Array	Editing in Component Parameter dialog box: A. Enter string values directly—Example: "Button 1" "Button 2" "Button 3"(each string is enclosed with double quote marks, spaces separate each string). B. Enter values from a file—Example: <filename. If the filename has no path specified, the workspace 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. 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. D. Enter value by reference—Example: @Y, where Y is a the name of a Variable or Expression for a string array. Parameter editing on schematic: Highlight parameter value on schematic and enter array value enclosed with double quote symbols.
Enumerated Type (with form specific to the component)	Editing in Component Parameter dialog box: 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. 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. 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. 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.

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.

All parameter types except Enumerated, are directly available to define parameters in a schematic design. To use the Enumerated type for a schematic design, you must edit the design AEL file and implement the AEL code for the desired Enumerated type. Examples of AEL for Enumerated types can be observed in files *$HPEESOF_DIR/adsptolemy/ael/*.adf*. Search for *create_form_set* which references items defined in *create_constant_form* lines. The first argument of the *create_form_set* is used in following *create_parm* lines and defines the enumeration for that parameter.

Parameter Editing

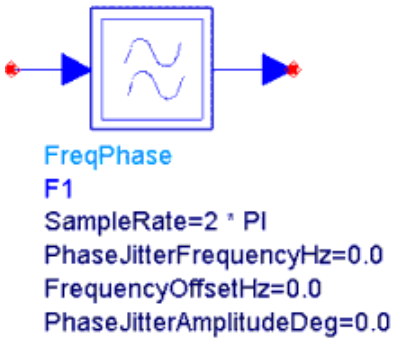
When an instance of a component or design is placed on the schematic, its parameters can be viewed below its symbol.

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.

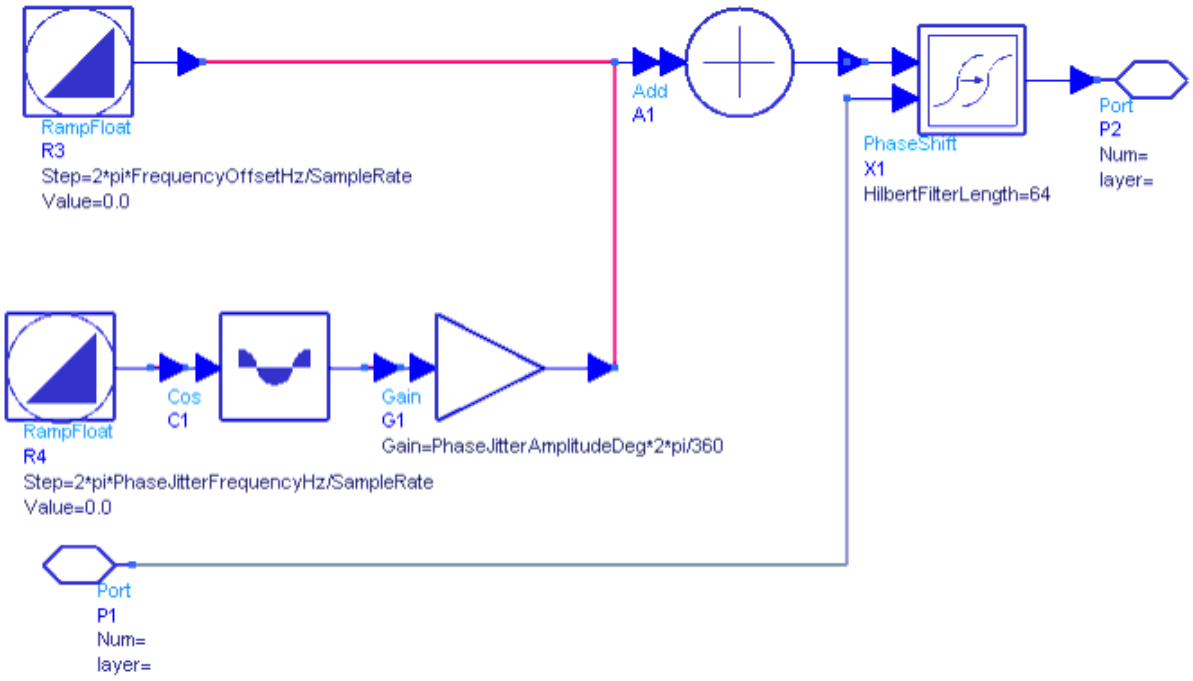
1. Choose **Insert** > **Component** > **Component Library**.
2. From the Library list, select **Numeric Communications** and then **FreqPhase** from the Components list on the right.
3. Place this component on the schematic.

Observe that its parameters (visible below its symbol on the schematic) are SampleRate, PhaseJitterFrequencyHz, FrequencyOffsetHz, and PhaseJitterAmplitudeDeg, as shown here.



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 *Simulator Expressions* (expsim), and *Measurement Expressions* (expmeas) documentation.

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. The FreqPhase schematic opens.



Notice that the various components in this schematic reference the top-level FreqPhase parameters by name. The RampFloat component Step parameter= 2*Π*FrequencyOffsetHz/SampleRate, where FrequencyOffsetHz and SampleRate values are passed in from the top level.

To view the parameters defined (or to define additional parameters) for this schematic design, choose *File > Design/Parameters*. The Design Parameters 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 *Creating Hierarchical Designs* (usrguide).

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 (real) 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 (real) expressions, to compute an integer parameter. In an integer parameter specification, all intermediate expressions will be calculated with double-precision floating-point (real) 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 (real) values, which may be numbers or expressions, and where j is the imaginary operator.

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

- An expression/function that returns a Cartesian form: complex (X, Y), where X is the real part and Y is the imaginary part.
- 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 fixed-point components used in ADS Ptolemy use 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, and finite precision bit format. The following describes several properties in common use by these components.

- 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-

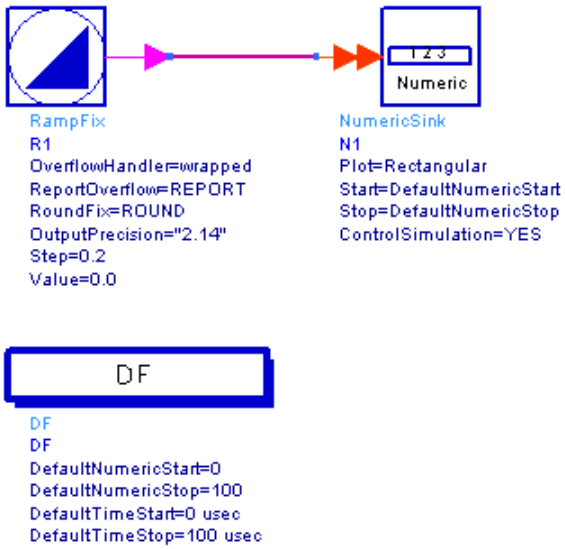
point scalar signals does not accept any matrix class signals, and vice versa. The automatic conversion from timed, complex or floating-point (real) signals to a fixed-point signal uses a default bit width of 32 bits with the minimum number of integer bits needed to represent the value. For example, the automatic conversion of the floating-point (real) 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* (ptolemy).

- ArithType is used to specify the arithmetic form of a fixed-point value. This parameter is an enumerated type with two options: *TWOS_COMPLEMENT* and *UN_SIGNED*. Fixed-point components in the Numeric Fixed Point DSP library use either arithmetic form; fixed-point components outside the Numeric Fixed Point DSP library use only the *TWOS_COMPLEMENT* form (the default).
  - For *UN_SIGNED* numbers there is no sign bit used; therefore a precision of 0/4 or 4.0 could have values from 0 to 15 inclusive.
  - For *SIGNED* numbers sign bit is included in the precision; therefore a precision of 0/4 or 4.0 could have values from -8 to 7 inclusive.
- RoundFix is used to specify the quantization property of a fixed-point value. This parameter is an enumerated type with two options: *ROUND* and *TRUNCATE*. The quantization property is used to convert a floating-point (real) 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 (real) value is used. For example, consider the floating-point (real) 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.
- OverflowHandler or OvflwType parameters are used to specify the overflow properties of fixed-point mathematical operations. These parameters are an enumerated type. The overflow parameter specifies the overflow characteristic to use when the result of a fixed-point operation cannot fit into the precision specified.

  OvflowType has two options: *WRAPPED* or *SATURATE*. OvflwType is used only by fixed-point components in the Numeric Fixed Point DSP library.

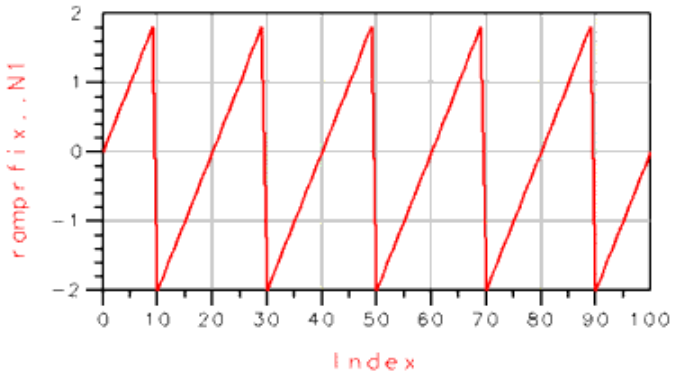
  OverflowHandler has four options: *wrapped, saturate, zero_saturate* , or *warning*. OverflowHandler is used by all fixed-point components except Numeric Fixed Point DSP components.

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.



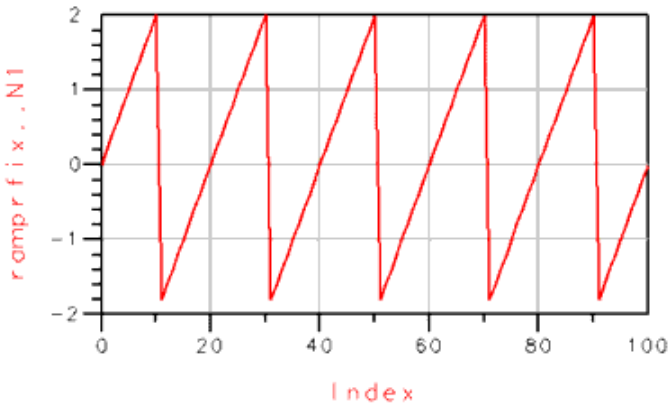
Schematic Using the RampFix Component

When OverflowHandler = *wrapped*, the following data display results:



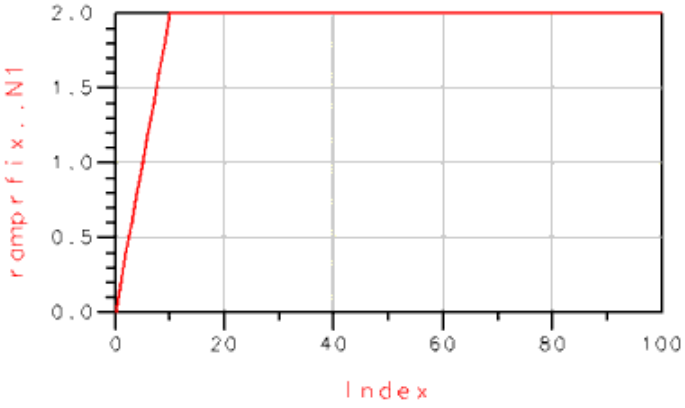
Simulation Plot with OverflowHandler Set to *wrapped*

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:



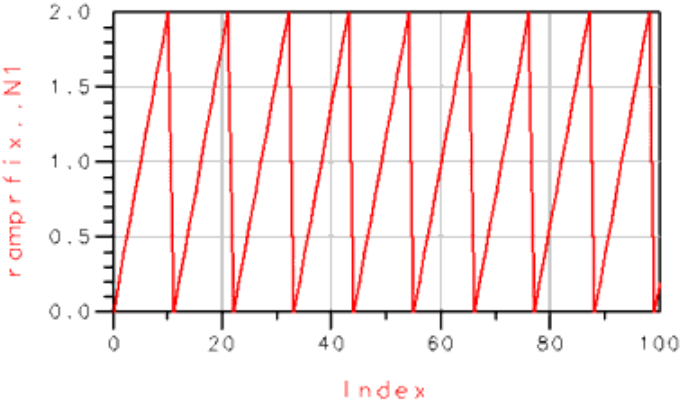
Simulation Plot with *TRUNCATE* Quantization

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:



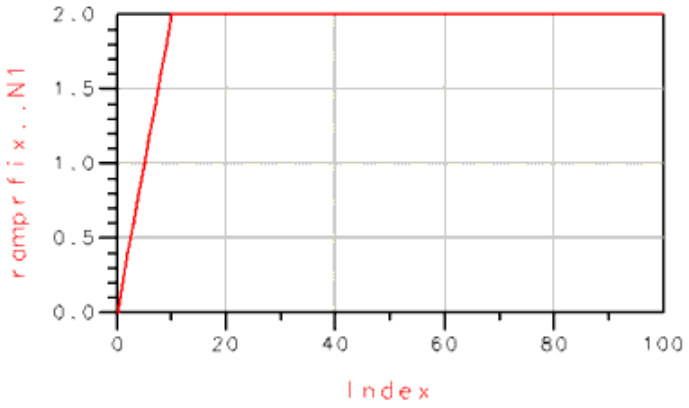
Simulation Plot with *TRUNCATE* Quantization and OverflowHandler Set to *saturate*

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:



Simulation Plot with *TRUNCATE* Quantization and OverflowHandler 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:



Simulation Plot with *TRUNCATE* Quantization and OverflowHandler Set to *warning*

Note that the saturate characteristic is used. Additionally, the warning mode results in ReportOverflow being set to *REPORT*, which reports the number of overflows at the end of the simulation.

- ReportOverflow is used to specify whether overflow is reported. This enumerated type parameter provides two options: *REPORT* and *DONT_REPORT*. Consider the preceding overflow displays; for each case, when ReportOverflow = *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 101 fixed-point calculations checked (8.9%)

  When you click this message, the RampFix component with the name R1 is highlighted in the schematic window.
  When ReportOverflow = *DONT_REPORT*, a warning message does not appear.
- UseArrivingPrecision is 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. This enumerated type parameter provides two options: NO or YES. UseArrivingPrecision 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 environment variables such as ~/, $HOME, $HPEESOF_DIR, or others. If no path name is provided, the current workspace data subdirectory is the assumed path for the file.

Array Parameters

There are two notations for defining array parameters: curly braces and double quotes. The curly brace notation is the preferred method for entering array values. It supports use of variables, variable expressions, simulator expressions, and multiplier symbols (p, n, u, m, k, M, G, ...). The double quote notation is deprecated and should only be used in the special cases described later in this section.

When defining arrays of integers, floating-point (real) numbers, complex numbers, or fixed-point numbers, the basic syntax is a simple comma separated list of values enclosed in curly braces, as shown in the following examples:

- Array of integer numbers specified with literal values
  - value entered in Component Parameter dialog box: @{1, 2, -6, 0, -3}
  - value entered on schematic: {1, 2, -6, 0, -3}
- Array of floating point or fixed-point numbers using a combination of literal values, multipliers, variables, variable expressions, and simulator expressions
  - value entered in Component Parameter dialog box: @{1.5, X, X+sin(Y), 2.3M, 1.8p, -4.1}
  - value entered on schematic: {1.5, X, X+sin(Y), 2.3M, 1.8p, -4.1}
- Array of complex numbers
  - value entered in Component Parameter dialog box: @{1.2+j*2.5, -2.3+j*1.3, 5.6-j*1.4, 2.8+j*log(Y)}
  - value entered on schematic: {1.2+j*2.5, -2.3+j*1.3, 5.6-j*1.4, 2.8+j*log(Y)}

The double quote notation uses a comma or space separated list of values enclosed in double quotes (double quotes are omitted when value is entered in Component Parameter dialog box). This notation does not readily support variables and variable expressions (variables and variable expressions need to be enclosed in parentheses in order to work), and does not support at all simulator expressions and multipliers (entering a simulator expression, e.g. 3*sin(X), will result in an initialization error, whereas using a multiplier, e.g. 3.5m, will result in the multiplier being ignored but no error or warning reported). Therefore, double quote notation should only be used in the special cases listed below:

- For arrays of strings (the curly brace notation does not support strings)
  - value entered in Component Parameter dialog box: "FOO" "BAR" "ABC"
  - value entered on schematic: """FOO"" ""BAR"" ""ABC"""
- For reading the array values from a file (the curly brace notation does not support reading values from a file)
  - value entered in Component Parameter dialog box: 1 -1 <file1.txt 2 -3
  - value entered on schematic: "1 -1 <file1.txt 2 -3"

(see Reading Array Parameter Values From Files for more details)

- When using a *repetition factor* for a certain value. When a value is repeated several times then the abbreviation value[n] can be used to represent n instances of the same value (the curly brace notation does not support repetition factors)
  - value entered in Component Parameter dialog box: 1[3] -1[5] (same as 1 1 1 -1 -1 -1 -1 -1)
  - value entered on schematic: "1[3] -1[5]" (same as "1 1 1 -1 -1 -1 -1 -1")
- When entering complex numbers using the format (real, imag) instead of real + j*imag (the curly brace notation *only* supports the format real + j*imag)
  - value entered in Component Parameter dialog box: (1.2, 3.5) (-1.8, 4.21) (6.12, -5.1)
    (same as (1.2+j*3.5) (-1.8+j*4.21) (6.12-j*5.1) or @{1.2+j*3.5, -1.8+j*4.21, 6.12-j*5.1})
  - value entered on schematic: "(1.2, 3.5) (-1.8, 4.21) (6.12, -5.1)"
    (same as "(1.2+j*3.5) (-1.8+j*4.21) (6.12-j*5.1)" or {1.2+j*3.5, -1.8+j*4.21, 6.12-j*5.1})

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
This syntax is only supported with the double quote notation (see Array Parameters for more details).

If the filename has no path specified, the workspace 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, fixed point, 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 can 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 manner similar to optimization attributes, there can also be parameters with swept attributes.

For more information on optimization in ADS Ptolemy, refer to *Using Nominal Optimization* (ptolemy). For more information on sweeping parameters in ADS Ptolemy, refer to *Performing Parameter Sweeps* (ptolemy).

Using Data Types

This topic reviews some basic material on Data Types that was introduced in *Data Types, Controllers, Sinks, and Components* (ptolemy), but then goes into more detail.

ADS Ptolemy uses different data types such as integer, fixed-point, floating-point (real), and complex in scalar or matrix forms. In ADS 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

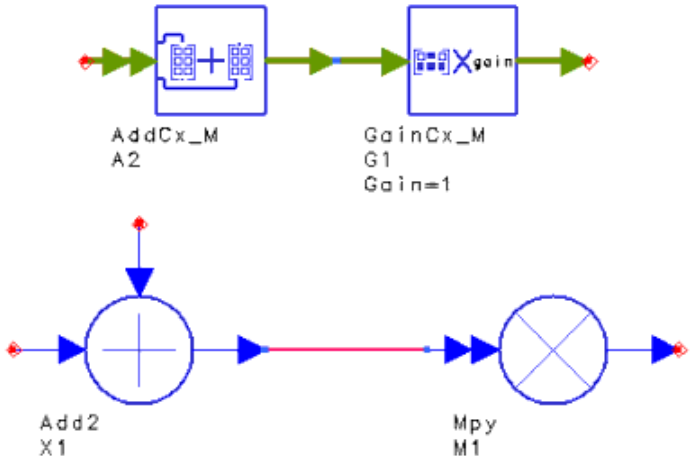
ADS Ptolemy schematics contain component stems with different colors and thicknesses. Each component input and output pin has an associated data type, and each type is represented in the component symbol by use of a color code and a thickness of stem. And, each component stem may have single or multiple arrowheads. The following table lists the data types.

Component Stem Color and Thickness

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

Stem Thickness

The following figure illustrates stem thickness:

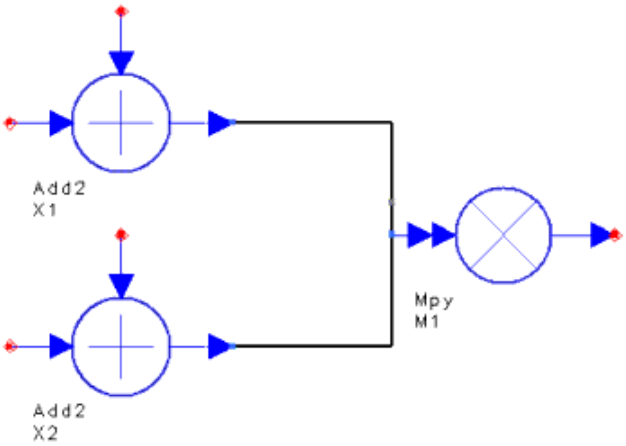


Matrix Data (Thick Lines) and Scalar Data (Thin Lines)

Single and Multiple Arrowheads

ADS 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. The following figure 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.



Single and Multiple Arrowheads

Data Types Defined

There are two general signal types in ADS 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 ADS 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
    twos complement (with sign bit included in x)
    unsigned
  - quantization type
    truncate, round
  - overflow type
    saturate, saturate to zero, wrapped
- floating-point (real) double precision floating-point (real) number
- complex pair of double precision floating-point (real) 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, floating-point (real), 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

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

{i(t), q(t), flavor, F *c* 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:

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

Conversion of Data Types

We introduced this topic in *Data Types, Controllers, Sinks, and Components* (ptolemy). In this section, we go into more detail. Most conversions do what you expect. For example, when converting from lower to higher precision data types such as integer to floating-point (real), no data is lost; only the format is changed.

When converting from higher to lower precision data types, such as floating-point (real) to integer, the outcome is determined by the 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 floating-point (real), floating-point (real) to complex, etc.) the obvious happens. For example, conversion from floating-point (real) 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 is followed that is generally simple and intuitive.

Numeric Scalar and Matrix Conversions

The following table outlines the rules regarding scalar conversions among numeric data types:

Numeric Scalar and Matrix Conversion Rules

From	**To**		
	Integer	*Fixed*	*Real*
Complex	round mag	round/truncate mag	mag
Real	round	round/truncate	
Fixed	round		

Note that *mag* 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 54 bits with the minimum number

of integer bits needed to represent the value in twos complement representation. For example, the integer 5 is converted to the fixed point number 0101.00000000000000000000000000000000000000000000000000 (precision "4.50"), whereas the floating-point (real) number 3.375 is converted to 011.01100000000000000000000000000000000000000000000000 (precision "3.51"). 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 (real) or Float (real) 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), flavor, Fc and t*}, the conversions between input and output of a converter are summarized below:

Timed To Float (Real)

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

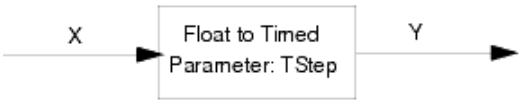
$y[n] = i(t)*\cos(2*\Pi*F_c *t)-q(t)*\sin(2*\Pi*F_c *t)$

when flavor = ComplexEnv

$y[n] = x(t)$

when flavor = BaseBand

Float (Real) To Timed



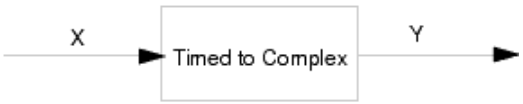
The FloatToTimed converter has one specific parameter, TStep. If x is the input and y is the output for this converter, then the y(t) packet has the following parts:

$i(t) = x[n*TStep]$

q(t) = 0.0
F_c = 0.0

flavor = BaseBand

Timed To Complex

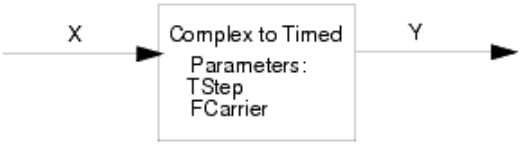


The Timed To Complex converter has no parameters. If x is the input and y is the output for this converter, then:

y[n] = i(t) + j*q(t) when flavor = ComplexEnv

y[n] = i(t) + j*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) = Real{x[n*TStep] }
q(t) = Imag{x[n*TStep] }
F_c = FCarrier

flavor = 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 numeric scalar, numeric matrix, and timed data categories discussed previously, consider the following:

- Conversion between numeric scalar and numeric matrix types is done by explicitly placing Pack and UnPack components. Automatic conversion is not performed between these two categories.
- Conversion between numeric scalar and timed data is done by placing the appropriate converters. Automatic conversion between these two categories is allowed.
- Direct conversion is not done between numeric matrix and timed data types.

The following figure illustrates the conversion among data types; timed data must be converted to numeric scalar before being converted to matrix.

Data Type Conversions

Automatic or Manual Data Type Conversion

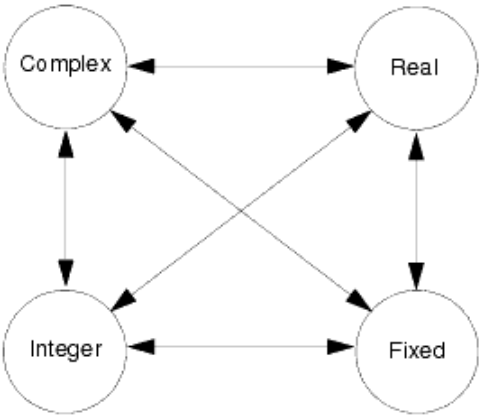
If the output of component A and the input of component B is the same (they are represented by the same color), data is simply copied from A to B. If the output of component A and the input of component B is different, conversion is needed.

Automatic conversion means that an appropriate converter is *spliced in* behind the scenes (not shown on the schematic). You may want to manually place an appropriate converter (from the Signal Converters library) in your schematic, which will be a visual conversion reminder and will help you decode any error messages.

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

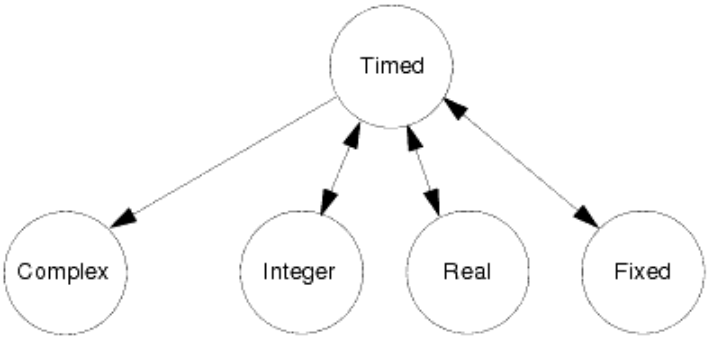
Allowed and Disallowed Automatic Conversions

Automatic conversion is available among all numeric scalar types. The same is true for matrix types. The following figure summarizes the allowed conversions.



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 illustrated in the following figure.



Automatic Conversion of Timed and Scalar Numeric Types

Automatic complex-to-timed conversion is *not* supported because carrier frequency information must be supplied by the user; for complex-to-timed conversion, place a ComplexToTimed converter and enter the appropriate parameters.

Automatic conversion of float (real) to timed, fixed to timed, integer to timed, or complex to timed must have at least one component in the design defining the TStep.

There is no automatic conversion between scalar and matrix data (or vice versa). In the Numeric Matrix Library, Pack_M, PackCx_M, PackFix_M, and PackInt_M are used to *pack* scalar data into matrix data; UnPk_M, UnPkCx_M, UnPkFix_M, and UnPkInt_M *unpack* the data (back to scalar). Place the converters where needed in your design. (Otherwise, 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.)

Using Nominal Optimization

This topic describes using nominal optimization with ADS Ptolemy. Nominal optimization, also called performance optimization, uses 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 Analog/RF Network simulators. Details regarding nominal optimization are provided in *Nominal Optimization* (optstat).

This topic discusses a DSP example on optimizing bit-width and describe information on signal processing parameter types to be aware of when performing optimization with ADS Ptolemy.

Optimizing Various Parameter Types

To optimize a real, integer, or fixed-point parameter type, the procedure is similar to standard nominal optimization.

To optimize complex, precision, array, string, or filename 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. This is because the simulator only sweeps numbers and these parameter types are strings that are interpreted by the simulator.

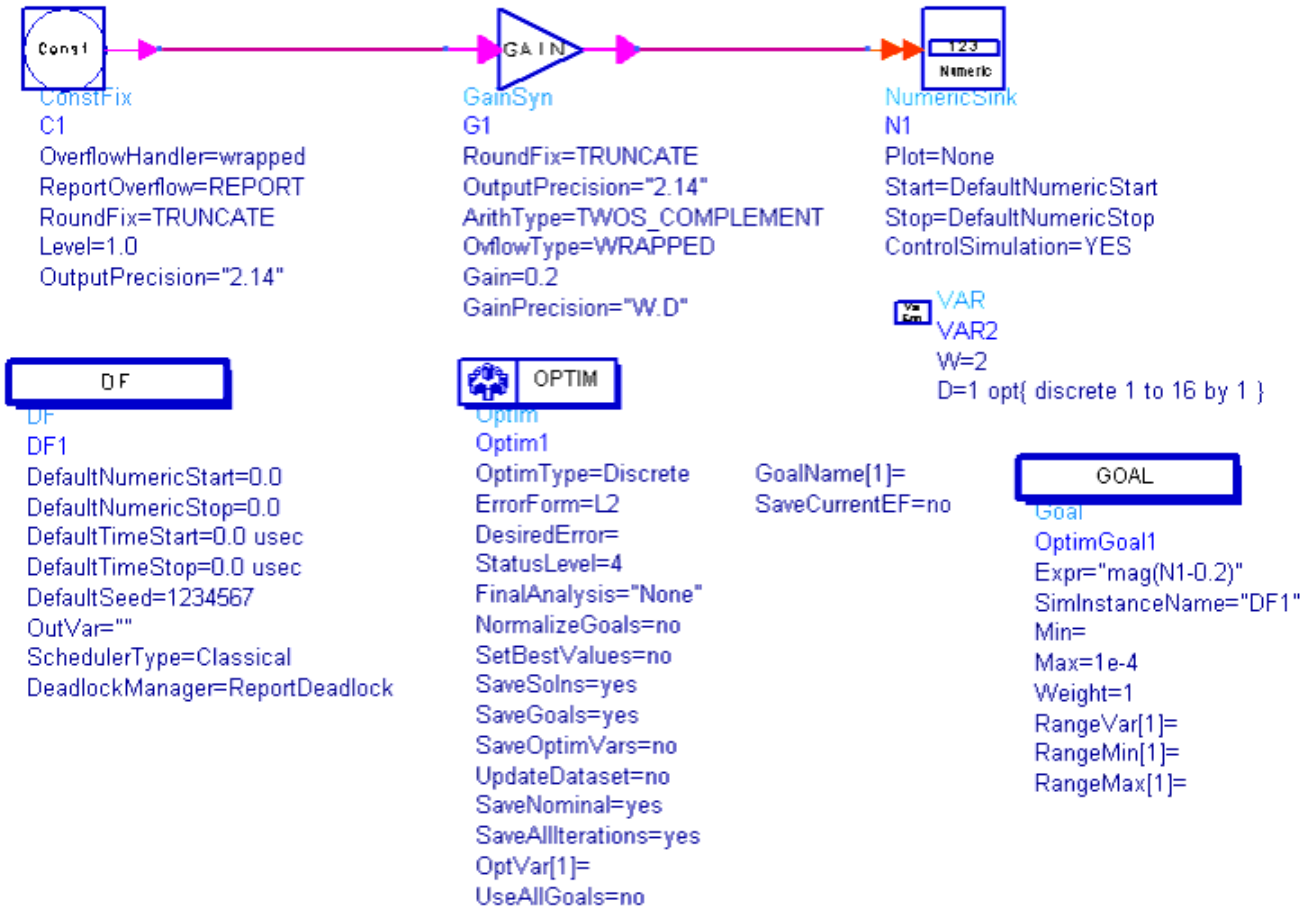
To reference an optimized variable for parameter types that use strings, the variable is defined in a VAR (Variables and equations) component with Standard entry mode and Optimization/Statistic Setup enabled. Once defined, this variable can be used as a component parameter. If you type this variable on the schematic, it must be enclosed in quotes " "; if you enter this variable 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. We will build a simple design as shown in the following figure. To help you set up this design, copy the workspace *dspopt_wrk* from the *examples/Tutorial* directory, and modify the design *simpleopt2* as described in this section.



Optimizing Input and Output Precision of a GainSyn Component

The value of the input and output precision of GainSyn in component G1 is optimized to achieve a *dc* data value of 0.2 in the sample stored in the numeric sink. The goal is to represent the gain with the minimum number of bits possible.

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 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.

The design consists of:

- A fixed constant output (ConstFix) source component (from the Numeric Sources library), with the Level parameter set to **1.0**.
- A GainSyn component, with **Gain=0.2** and **GainPrecision="W.D"**.
- A NumericSink, with default values accepted.
- A DataFlow controller, with default values accepted, except set DefaultNumericStop to **0.0**.
- A VAR component, set up as described in the following paragraphs.
- An Optim controller, set up as described in the section *Setting Optimization Job Parameters* (optstat). Use the default values, except set **MaxIters = 30**. The default

optimizer type is the **Random** optimizer.
- A Goal component, for the expression **mag(N1-0.2)**.

Set Up the VAR Component

To set up the VAR component, first see the following figure.

VAR Component Parameters

Double-click the VAR symbol in your schematic to display its dialog box. The following describes each parameter that must be set so the VAR component matches what is shown in the previous figure.

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.

   Note
   The W and D labels are user-defined variable names.
3. Choose the **Tune/Opt/Stat/DOE Setup** button.
4. From the Optimization Status dropdown menu, select **Enabled**.
5. From the Type dropdown menu, 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**.

   Note
   Steps 5 through 8 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**.

Note that the Weight parameter weighs the importance of one goal to the other goal(s). Generally, the first goal may be more important, for example 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, the Weight of the first goal is set to 1e9.

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) are the same as for any optimization. To review these procedures, refer to *Specifying Component Parameters for Optimization* (optstat).

Wireless Test Bench Designs

Using Ptolemy in ADS, a system designer can create a DSP system design that is available to an RFIC designer as a wireless test bench (WTB) model. The wireless test bench is a mechanism by which a system designer can make system measurements available to an RFIC designer. The RFIC designer can then validate and verify the RFIC design or device under test (DUT).

Creating a Wireless Test Bench Design

Creating wireless test bench designs using Ptolemy components in ADS is similar to setting up other Analog/RF designs for cosimulation. The design must be created in a DSP schematic using only DSP components. In the *File > Design Parameters* select simulation model *Wireless Test Bench* .

The WTB design can be split between two parts.

- The first part of the WTB design is the group of components that generate a signal appropriate for stimulating the RFIC or DUT. Ports (appropriately named) added at the output of this part of the design will act as WTB model output and DUT input.
- The second part of the WTB design is the group of components that process the DUT output and perform various system measurements using Ptolemy Sink components. Ports (appropriately named) added at the input of this part of the design act as WTB model input and DUT output.

The input ports must have an EnvOutSelector or EnvOutShort (from the Circuit Cosimulation library) to select the correct frequency to be extracted from the Circuit Envelope analysis of the DUT for system measurement. Any port that is not connected to an EnvOutSelector or EnvOutShort is considered to be an output port.

A DF controller (from the Controller library) must be added to the design.

Create the WTB design parameters that the RFIC designer can use to modify the WTB design's configuration, using the *File > Design Parameters* in the ADS schematic window.

Wireless Test Bench Design Examples

For examples on how to create a WTB design, look at the designs listed with the example workspaces in the following table. Each design has instances of the pre-configured WTB model components. Push into the components to view the design.

These workspaces are contained in the 3GPP W-CDMA. WLAN, and TD-SCDMA Design Libraries and can be installed by using the custom installation option.

Wireless Test Bench Design Examples

Example Workspace	License Required
TDSCDMA_RF_Verification_wrk	mdl_tdscdma
WCDMA3G_RF_Verification_wrk	mdl_wcdma3g
WLAN_RF_Verification_wrk	mdl_wlan
UWB_RF_Verification_wrk	mdl_ultrawideband

Note
RangeCheck components used in the pre-configured WTB designs are created specifically for those WTB designs. Do not use the *RangeCheck* components when you create your own WTB designs.

Setting the Units for WTB Design Parameters

The system designer can use the list of units provided in the Design Parameter dialog box to correctly specify the unit used for a parameter. If the parameter value is a unit that is not part of the list, such as *percent* , and the system designer wants to communicate the units to the RFIC designer, then the unit can be placed within parentheses at the end of the parameter's description text.

Categorizing WTB Design Parameters

WTB designs support a feature that enables a person using the WTB design to categorize the parameters. It is helpful for the System designer to group the parameters in a way that makes it easier for the RFIC designer.

To categorize the list of parameters, define a string type of parameter at the beginning of the group of parameters that belong in a category. The parameter must have a description field specified that contains a very short description of the category. The parameter's default value must be set to *Category* to identify this parameter as a categorization parameter. And, the parameter's attributes must be set such that it will not be netlisted and not displayed.

Information Parameters

The system designer can insert various information parameters that describe a particular aspect of the WTB design, such as instructions related to timing setup. To create such a parameter, define a string type of parameter and set its attribute such that it will not be netlisted and will not be displayed on the schematic. Leave the parameter's default value blank.

Using a WTB Design in ADS

To use a WTB design in ADS:

1. Create a DUT in ADS.
2. Open a new Analog/RF schematic window.
3. Add an instance of the WTB design and the instance of the DUT; connect them.
4. Add an Envelope controller.

5. Simulate this design and generate the dataset to view the results.

Creating a Results Display for WTB Designs

A WTB design can have complex data that the RFIC designer will not know how to interpret. To help simplify data analysis, the System designer must use the following steps:

1. System designer must use a simple DUT in ADS to create a dataset.
2. Open a new DDS window. Add one or more new pages and name them appropriately.
3. Leave *page 1* untouched and blank; this page is used for other Analog/RF automatic plots.
4. Add correct equations/plots/configurations on the new pages.
5. Save this DDS file as a template by choosing *File > Save as Template* on the DDS window. Select the *User* category to save the template.
6. The template file is saved under *$HOME/hpeesof/circuit/templates*.
7. Copy the template file to *_wrk/adsptolemy/templates*.
8. Place an *OutputOption* controller in the WTB design's schematic window. (The *OutputOption* component can be placed at any level of hierarchy in the WTB design.)
9. Add the data display template name, created above, to the list of names. More than one data display template is allowed for a given WTB design.
10. Re-export the WTB design.

Now, when this WTB design is used in a simulation, the templates named in the *OutputOption* controller will be inserted in the DDS window that appears at the end of the simulation.

Circuit Envelope Parameters

Some parameters must be defined as WTB design parameters to ensure compliance with Circuit Envelope requirements within the WTB design's simulation setup.

- **CE_TimeStep** The WTB design must have a *CE_TimeStep* parameter that enables the RFIC designer to set the Circuit Envelope time step. If this parameter is not added to a WTB design, the time step will always be set to 0.01 µsec.
- **FSource** The WTB design can optionally have an *FSource* parameter to enable the RFIC designer to set the source carrier frequency.
- **FMeasurement** The WTB design can optionally have an *FMeasurement* parameter to enable the RFIC designer to set the measurement carrier frequency.