Agilent Technologies

SystemVue 2011.03
2011
Users Guide

This is the default Notice page

Acknowledgments Mentor Graphics is a trademark of Mentor Graphics Corporation in the U.S. and other countries. Microsoft®, Windows®, MS Windows®, Windows NT®, and MS-DOS® are U.S. registered trademarks of Microsoft Corporation. Pentium® is a U.S. registered trademark of Intel Corporation. PostScript® and Acrobat® are trademarks of Adobe Systems Incorporated. UNIX® is a registered trademark of the Open Group. Java™ is a U.S. trademark of Sun Microsystems, Inc. SystemC® is a registered trademark of Open SystemC Initiative, Inc. in the United States and other countries and is used with permission. MATLAB® is a U.S. registered trademark of The Math Works, Inc.. HiSIM2 source code, and all copyrights, trade secrets or other intellectual property rights in and to the source code in its entirety, is owned by Hiroshima University and STARC.

Errata The SystemVue 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.

The SystemVue Environment

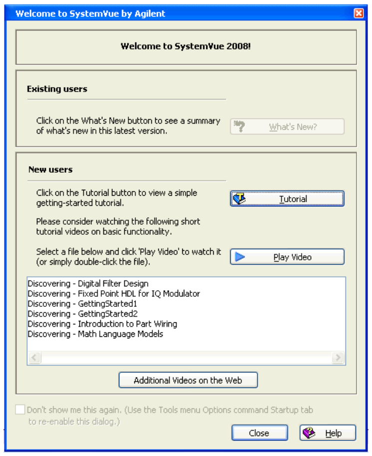
This section will familiarize you with the user interface of SystemVue. These sections will include introductions of various components of the SystemVue software screen. To get more detailed information about using these features, read *Using SystemVue* (users) section.

Contents

- *Starting SystemVue* (users)
- *Getting Started Dialog Box* (users)
- *Design Environment* (users)

Starting SystemVue

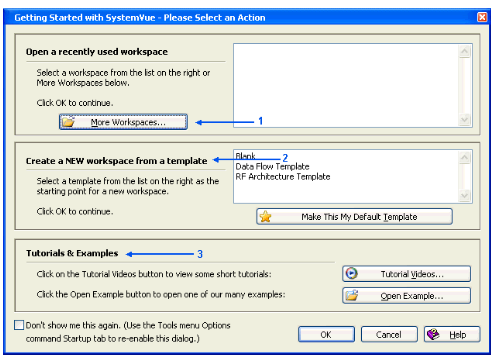
To start SystemVue:
Click **Start > SystemVue**. Loading SystemVue screen opens. After a while SystemVue main window opens launching the welcome dialog:



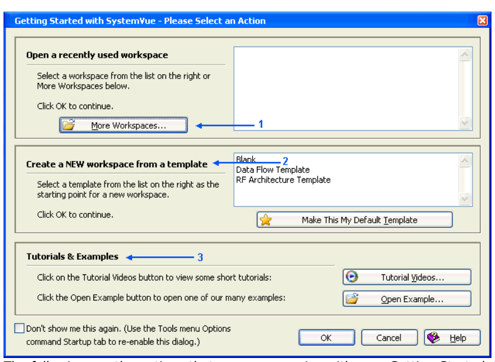
This window is also called the **splash screen**.

> **Note**
> You might want to view some of the videos listed in the **New Users** section, even if you're an experienced user. They are typically short and cover some of the most useful convenience and quick-start topics.

To close the window, click **Close**. The control automatically shifts to the **Getting started** screen with the title - **Getting Started with SystemVue - Please Select an Action**.



Getting Started with SystemVue Dialog



The following are the options that you can exercise with your Getting Started dialog box:

1. **Open a recently used workspace**
   You can select a recently used workspace from the list on the right. If the recent space is not listed, click **More Workspaces** to locate a workspace on your computer system.
2. **Create a NEW Workspace from a template**
   You can select a template fr0om the list on the right. Select **Default** to start withe the default template or you can select any template and click **Make This My Default Template** to make the selected template as default one.
3. **Tutorials and Examples**
   There are two buttons available to access help for videos and examples.

There is "Don't show me again" button on this page as well, but it is recommended not to select it (for new users).

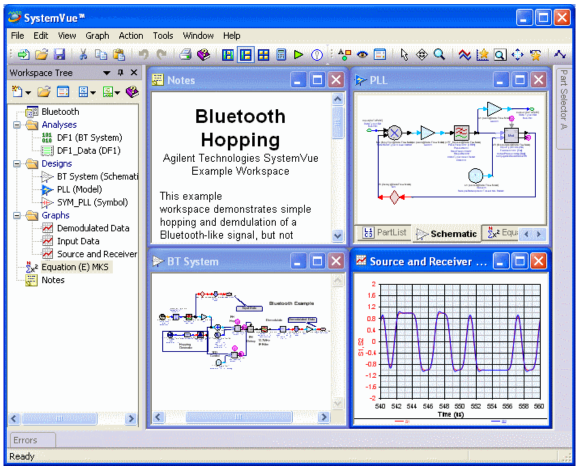
If you simply close this window, you get to the default workspace.

Note
Click the Start Page button , (the first button in the main toolbar) to open this dialog anytime.

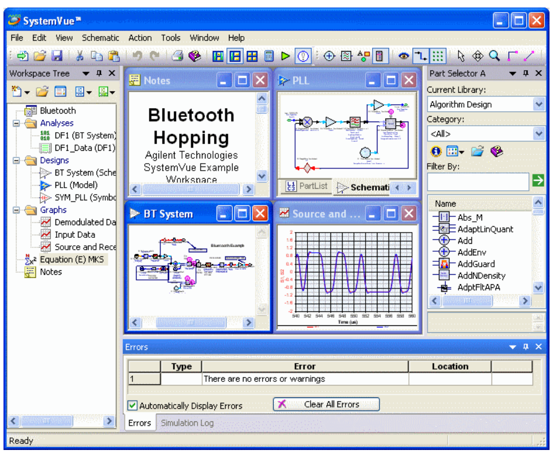
SystemVue Design Environment (User Interface)

The SystemVue design environment consists of menus, windows, toolbars, and standard editing options. It is easily integrated with other programs, and you can use it to view multiple projects, schematics, and simulations at the same time.

The environment is versatile. It can look like this...

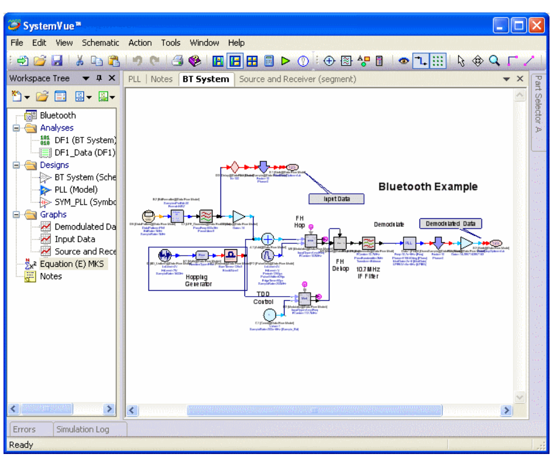


or like this...



We show you the second environment by **default** so you know what is available and what it looks like. In the design area, an un-maximized window can be re-sized by clicking on a boundary and dragging it. Three buttons on the right of the title bar control a design window's viewability. Click on to minimize the window. Click on to maximize the window. Click on to remove the window.

Also, it's easy to switch to a tabbed window view, by using the *Window Menu* (users). Note the view window tabs at the top of the graph:



The default SystemVue design environment consists of the following:

- ***Menu* (users)** – Contains all of the commands used in SystemVue.
- ***Toolbars* (users)** – Contains buttons that are shortcuts for commonly used commands.
- ***Workspace Tree* (users)** – Displays a hierarchical list of items in your project.
- ***Part Selector* (users)** – Lists the electrical parts in a specific library.
- ***Library Selector* (users)** – Lists all of the designs in a specific library.
- ***Tune Window* (users)** – Contains settings that let you modify variables for a circuit in your design.
- ***Simulation Status Window* (users)** – Displays the status of the running simulation.

- ***Error Log* (users)** – Displays error information.
- ***Status Bar* (users)** – Displays useful information at the bottom of the SystemVue window.
- ***Design Windows* (users)** – where all the real work takes place, are placed within the gray workspace area.
- ***Simulation Log* (users)** – Information regarding any running simulation.

Click on the page to read more.

Design Windows

All working windows in the workspace area are called **Design Windows**.

To show or hide any of the windows:

- Click **View** on the SystemVue menu and select the window.

As you click inside each design window, the toolbar for that window will appear and may replace toolbars from the previous design window. You can move or dock toolbars anywhere in SystemVue by grabbing the bar on the left and moving it (if docked) or grabbing the title bar (if floating). If a design window is active (selected) among several windows, its title bar becomes dark blue. The above examples show different toolbars to the left of the main toolbar.

If you re-size or maximize a window, the contents will grow (or shrink) adjusting to the new window size. Notes will reformat. Graphs print full-page and schematics print according to their defined physical sizes (using shrink to fit if the page will not fit on the paper).

Design windows are special in that they can show multiple views of itself, so you can see the partlist in one tab, the schematic in another and the layout in yet a third tab. Right click on the window and select the tab option to get another view of the design.

Workspace Area

The background of this area is gray and it holds any windows that are opened i.e. schematic window, graph, notes window,smith chart etc.

You can open as many windows as you want in this area. The default schematic menu icons are listed because that is the default window opened for any new workspace. This menu icons list is changed with the window that is selected. If you open up a graph window, the "schematics icons list" will give way to the "graph icons list".

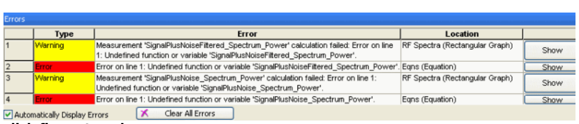
Special Operations

Several operations that you can work out on the windows in your workspace area:

1. Tile them vertically or horizontally
2. Close all the open windows
3. Make the windows opened as **tabbed**
4. Show/Hide all other docking windows (so that only the opened window is visible)
5. Show all output windows

Error Log

The Error Log displays near the bottom of the SystemVue window and alerts you to potential problems in your design. You can display the Error Log whenever you open SystemVue. Or, you can have SystemVue display the Errors window only when higher-level error messages are generated.



click figure to enlarge

To open or close the Error Log:
1. Click **View** on the SystemVue menu and select **Error Log**, *or*
2. If the window is open, click the close button (the x on the upper left) of the Error Log to close the window, *or*
3. Click the Errors Button in the main toolbar.

To automatically display the Error Log for higher-level errors:
1. Click the **Automatically Display Errors** check box.

To clear out the messages in the error window
1. Click the **Clear All Errors** button.

Reviewing Error MessagesThe Error Log displays informational, warning, error, and critical messages. The messages are color-coded by message type.

- **Informational Message** – Green indicate a potential problem in your design.
- **Warning Message** – Yellow indicate a minor problem in your design.
- **Error Message** – Red indicate a problem in your design.
- **Critical Message** – Black indicate a critical problem in your design.

Messages always have a Show button. Click to bring up a schematic showing the highlighted error or a dialog showing the error line. If you have an undefined error, the Show button may do nothing.

More than one error message can come from the same part. Look at the last error in the list (the first to get thrown) to view the root error.

An instantiation error in a model during a simulation usually means that a parameter was bad (invalid or out of range). It might also imply the model couldn't be found or has changed since it was last used.

Using the Errors Window Button

You can also check for messages by viewing the Errors Window button on the SystemVue toolbar. The color and image on the Errors Window button show the highest-level message in the Error Log.

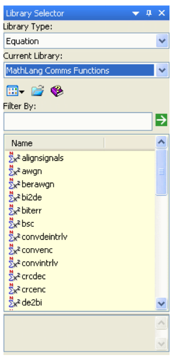
Button	Symbol	Meaning
White		Indicates there is no message.
Green		Indicates an information message.
Yellow		Indicates a warning message.
Red		Indicates an error message.
Black		Indicates a critical message.

Library Selector

The Library Selector is a toolbar that gives you quick access to libraries of archived workspace items(datasets, designs, equations, etc.). The light-yellow background of the Library Selector distinguishes it from the Part Selector and other toolbars. It is used to display libraries of item types such as datasets, designs, or equations.

A library is a collection of one type of object found in SystemVue. For example, a library of equations contains a collection of Equation Blocks, and only other Equations Blocks can be added to this library of equations. Schematic, Models, and Symbols are all considered to be a design, and so a library of designs can contain all three of these. Libraries of parts cannot be displayed in the Library Selector and must be viewed in the Part Selector.

The Library Selector operates for other objects much like the Part Selector operates for parts. The Library Type sets the type of object library you want to view and the Current Library sets to the particular library you want to display. The Filter By feature can be used to display a subset of the Current Library. When you select an item, detailed information about it is displayed in the information window at the bottom.



To edit a workspace item:
1. Double-click an item in the Library Selector list. The object is placed into your workspace and available for editing. Note that if this is a model or symbol you are currently using in your workspace then the in-workspace version of the model/symbol will override the library version.

> **Hint**
> This is a great way to send self-contained workspaces to your coworkers, by embedding any custom models or symbols (or vendor models or symbols) into the workspace itself.

To set the library type:
1. Click the Library Type pulldown and select a library type to find.

To change libraries:
1. Click the Current Library pulldown and select a library to switch to. The Current Library pulldown only contains libraries of the type set in Library Type.

To search for specific objects:
1. Type the text for the items you want in the **Filter By** box. For example, in the library shown above type **rand** to find the randint functionor any objects whose name or description contains that text. The filter is applied to the part name and description.
2. Click the Go button ( ) to display the updated list.

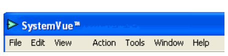
To copy an object into a library
1. Right-click the item in the workspace tree and select the **Copy To** menu to copy the object to a library.

To change which columns are displayed:
1. Right-click the column heading and check on or off the columns you want to see. Click a column heading to sort by that column.

To change the way the selector shows parts:
1. Right-click the white area in the selector and select from the View submenu.

Menus

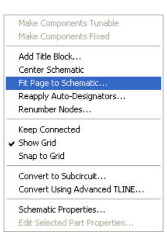
The SystemVue menus are located on the menu bar at the top of the SystemVue window. There are several menus that appear automatically whenever SystemVue is started. These are called default menus.



The other SystemVue menus are called object menus. They are specific to the windows in a design and appear only when that window is active. For example, the Schematic menu is visible only when the Schematic window is active.

The top is the main SystemVue menu like the Windows menu. It contains the basic menus alongwith a **Schematic** menu which is dedicated to SystemVue schematic.

The schematic menu is used for modifying the schematics in the workspace.



List of common icons

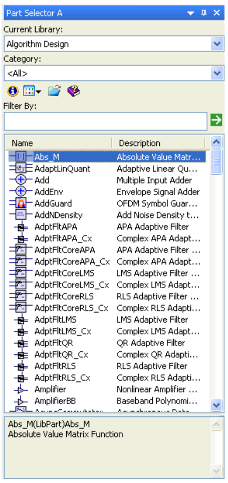
Most of these are common Windows icons itself.

For more information about all of the SystemVue menus, see *Appendix B Menus* (users).

Part Selector

The Part Selector is a toolbar that lets you add **parts** to a design. It displays a list of parts from the currently selected library. A library is a collection of objects that can be used in SystemVue. The Part Selector only displays libraries of parts. The Library Selector is used to display libraries of other types. Algorithm Design is the default library. You can use the Category and Filter By features to display a subset of parts from the current library. When you select a part, detailed information about it is displayed in the information window at the bottom.

SystemVue provides two part selectors: A and B. Part Selector A is the default, but you can display both part selectors at the same time. The options for viewing either part selector are found in the *View Menu* (users). Having both Part Selectors open lets you work with two libraries at once. Building a custom library of parts is easier with both Part Selectors open, because you can set one to view the custom library of parts as you build it.



You can use the Part Selector toolbar to perform the following tasks.

Click this button	To do this
	Get reference information for the currently selected part.
	Select options to change the way parts display in the Part Selector window.
	Manage the part libraries. Click this button to open the Library Manager.
	Get online Help.

To place a part:
1. Click a part in the Part Selector list. Notice the part details that display in the information window.
2. Click in the Schematic window to place the part.

To view a subset of a part library:
1. Select a subset of parts to view from the Category list.

Note
The **All** category displays all available parts in the selected library.

To change part libraries:
1. Click the *Current Library* pull-down and select a library name to display all of the parts in that library.

To add a part library:
1. Click the Library Manager button ( ) and select Library Manager to get a dialog allowing you to add existing libraries.

To search for specific parts:
1. Type the text for the parts you want in the **Filter By** box. For example, type **math** to find the math function part or any parts whose names or descriptions contains that text.
2. Click the Go button ( ) to display the parts in the Part Selector window.

To copy parts to a library:
1. Right-click the part you want to copy, and then select the name of a library from the Copy To menu. A copy of the part is automatically placed in the new library.

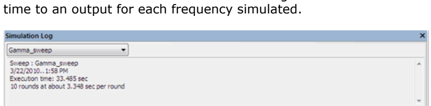
To change which columns are displayed:
1. Right-click the column heading and check on or off the columns you want to see. Click a column heading to sort by that column.

To change the way the selector shows parts:
1. Right-click the white area in the selector and select from the View sub-menu.

Simulation Log

By default, the Simulation Log shows near the bottom of the SystemVue window. However, it is a docking window that can float or be docked in the SystemVue window.

The content of the simulation log will depend on the simulation or evaluation that is run. Each will show different information that ranges from a date and time run with execution time to an output for each frequency simulated.



To open or close the Simulation Log:
Click View on the SystemVue menu and select Simulation Log

OR

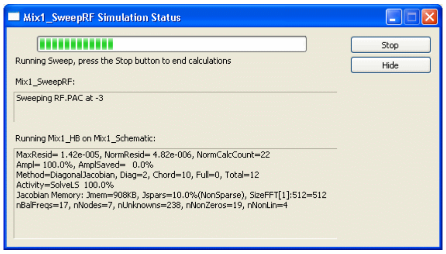
If the window is open click the close button (the X on the upper left) of the Simulation Log to close the window.

To select the analysis or evaluation you want to view:
1. Click the pull-down and select the desired analysis or evaluation.

Simulation Status Window

When a simulation is running, various output will be shown in this window, including the type of simulation being run and the status of the simulation. You can press the Stop button to stop the calculations at anytime. You can also press the Hide button to hide the status window, this will hide the status window and continue running the simulation. The details of the active simulation are shown in the main box of the simulation status window.

An example sweep



Click the Stop button to stop the simulation run.

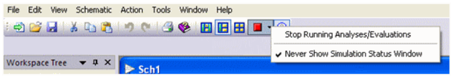
Click the Hide button to hide the status window but continue the simulation. There is also a Global option that always hides the status window. See the global options section.

Hiding the Simulation Status Window

The Hide button on the Simulation Status Window allows you to hide the currently running simulation's status window. There is also a global option that can be set to never show the simulation status window. There are two ways to turn the "Never Show Simulation Status Window" option on or off:

- Use the Global Options Page. See *General Global Options* (users) for information on making the setting this way.
- Use the toolbar start or stop button drop down.

To set the "Never Show Simulation Status Window" option from the toolbar:

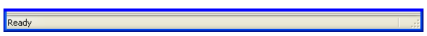


1. Click the drop down arrow beside the start or stop button on the main toolbar. The stop button will only be visible when a simulation is running.
2. Toggle the "Never Show Simulation Status Window" menu entry.

When toggled on from the toolbar when a simulation is running the status window will immediately be shown.

Using the Status Bar

The status bar is located at the bottom of the SystemVue window.



It spans the width of the window and contains useful information or messages regarding your current task. If there is no information, the default message is **Ready**. When an action successfully completes, the default message is **Done**.

You should read the information in the status bar on a regular basis for assistance in using the program.

Toolbars

There are many toolbars in SystemVue. The main SystemVue toolbar is referred to as a *default toolbar* (users). The main SystemVue toolbar is shown below:

SystemVue also has a number of other toolbars called object toolbars. They are specific to the windows in a design and appear only when that window is active. For example, the Schematic toolbar is visible only when the Schematic window is active.

To reposition a toolbar:

- Drag the toolbar to the new location.

To re-size a toolbar:

- Drag a corner of the toolbar until it changes to a different size.

To create a floating toolbar:

- Drag the toolbar to the desktop.

> **Note**
> If you do not want the toolbar to dock to the sides or top of the SystemVue window, hold down the CTRL key while dragging.

Using a Default Toolbar

You can use the main SystemVue toolbar to perform basic editing commands, such as opening, saving, or printing designs.

To show or hide a default toolbar:

- Click **View** on the SystemVue menu and select the toolbar you want to show or hide from the Toolbars menu.

> **Note**
> Toolbars that are currently open have a check mark next to them.

To display the default toolbars on startup:

1. Click **Tools** on the SystemVue menu and select **Options**.
2. Click the **Startup** tab.
3. Click the **Use Default Toolbar Settings on Startup** button.
4. Click **OK**.

Using an Object Toolbar

The object toolbars let you perform actions for specific windows.

To show or hide an object toolbar:

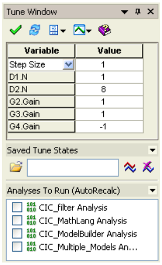
- Click **View** on the SystemVue menu and select either **Show All Object Toolbars** or

Hide All Object Toolbars from the Toolbars menu.

For more information about all of the SystemVue toolbars, see *Appendix C Toolbars* (users).

Tune Window

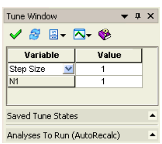
It is a tool used for *Tuning Variables* (users). It is one of the most powerful features of SystemVue. You can use tuned variables(real time) almost anywhere in SystemVue, including part parameters.



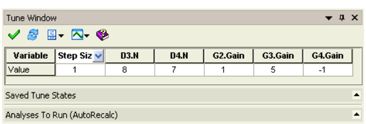
Any numeric parameter in a part can be made tunable. To get more information on how to tune variables, see *Tuning Variables* (users).

Tune Window Component	Purpose
Accept Tuned Settings	Applies the current Tune settings to the graphs, etc.
Refresh	Scans for currently tunable variables
Variable Options	Sets Tune Window Variable settings
	• Hide Name Prefix - Omits the name prefix, so the name is as short as possible (overrides Long Names too). Duplicate variable names are common in this mode, which is confusing, so the recommended setting is OFF.
	• Long Names - Display the full name of the tunable variables
	• Select Variables - Displays a window which allows several variables to be selected at once.
Graph Checkpoints	Enables graph checkpoints
Help	Brings up help on tuning. (This page of documentation)
Variable Grid	Contains the tunable variables
	Variable Tuning Mode (dropdown)
	• **Normal** - tune (increment / decrement) by a percentage value, usually 5 or 10%
	• **Step Size** - tune by adding or subtracting the step size
	• **Standard** - Use Standard part values. (Limits tuning to specified "standard" values, which is useful for physical "lumped" parts i.e. resistors, capacitors, etc.)
Tuning Value	Amount to tune variables by (in conjunction with tuning mode)
	Variable - The name of a tunable variable, with optional info as set by the Item Menu above.
	Value - The value of a tunable variable. Click grid cell to activate tuning this variable.
Saved Tune States	Caches the current variable settings
	• Use These Settings - Opens saved settings
	• Settings Name - Name of the current settings
	• Checkpoint the Graphs - Places checkpoint traces on the graphs
	• Remove All Graph Checkpoints - Removes checkpoint traces from all the graphs (but does not delete named settings)
Analysis To Run (AutoRecalc)	Provides easy access to the Automatic Recalc settings of all the Analysis in your workspace
	• **Check** an analysis to enable its AutoRecalc mode, so that the analysis will run when a variable is tuned
	• **Uncheck** an analysis to disable its AutoRecalc setting, so the analysis will NOT automatically run

The Tune Window is collapsible so as to reduce screen clutter, the **Saved Tune States** and **Analysis To Run** panels can be hidden, via the "Fold" button on the right of each panels titlebar. (Click the "Unfold" button to restore the panels to full height.)



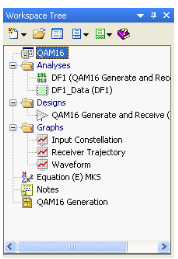
The Tune Window also has a horizontal display mode, which is automatically triggered when the Tune Window is wide:



If you are tuning more than one variable which have the same name, you may notice duplicate names in the list if you have selected the option to "Hide Variable Prefix" which shows shortened variable names. The "Hide Variable Prefix" option is available by clicking the **Variable Options** toolbar button.

Workspace Tree

The SystemVue Workspace Tree displays a hierarchical list of items in your project such as designs, analysis, data sets, and graphs. With it, you can add, delete, or rename items. To use an item right-click the item and select from the menu or click and highlight the item and then click the item menu button shown below.



You can use the Workspace Tree toolbar to perform the following tasks:

Click this button	To do this
	Add a new item such as an analysis, design, or graph. Or, add an item from a library.
	Open the currently selected item.
	Open the properties window for the currently selected item.
	Pull down the menu of the currently selected item.
	Pull down the Workspace Tree menu to adjust the Tree appearance by letting you show/hide datasets, change the sorting order, show additional information, etc.
	Get Help.

To add an item to the Workspace Tree:

1. Click the New Item button ( ) and select the item you want to add.
2. Type a name in the Name box.
3. Type a description in the Description box, if any.
4. Enter any other parameters in the properties window.
5. Click **OK**.

To delete an item from the Workspace Tree:

1. Right-click the item you want to delete and select **Delete** from the menu.
2. Click **Yes**.

To rename an item in the Workspace Tree:

1. Right-click the item you want to rename and select **Rename** from the menu.
2. Delete the current name, and then type a new name in the box.
3. Click **OK**.

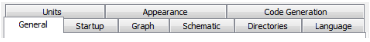
or slow double-click and type then click elsewhere when done

To copy an item to a library:

1. Right-click the item and select the Copy To sub-menu. Pick a library to copy to or use New Library to create a new library.

Setting Global Options for SystemVue

Customize your working environment to best suit your needs using the global application options. You can set options for things such as how numbers are formatted for, which windows to display at startup, and which directories to use. The global options are saved when the application ends and are restored the next time you run it.

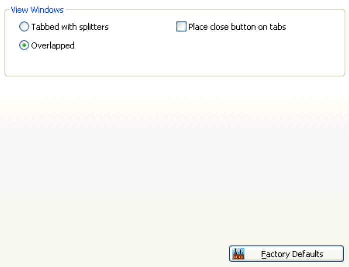


To set Global Options

1. Click **Tools** on the SystemVue menu and select **Options**.
2. Click any of the following option tabs:
   General (users)
   Startup (users)
   Graph (users)
   Schematic (users)
   Directories (users)
   Language (users)
   Default Units (users)
   Appearance (users)
   Code Generation (users)
3. Select the options you want.
4. Click **OK**.

Appearance Options Tab

Use the appearance options window to see the default directory paths.

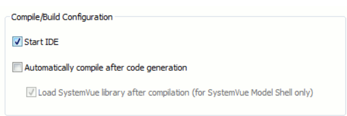


To change the appearance global options:

1. Click **Tools** on the menu and select **Options**.
2. Click the **Appearance** tab.
3. Adjust the settings:
   - **Tabbed with splitters** – Specifies the use of tabbed view windows, with splitter bars.
   - **Overlapped** – Specifies the use of multiple document interface (MDI) overlapping windows.
   - **Place close button on tabs** – When checked, the tab close button will be placed on the tab button itself (instead of being placed on the right).
   - **Factory Defaults** – Restores the original factory values to these settings.
4. Click **OK**.

Code Generation Options Tab

Use the Code Generation options behavior of Code Generation paths.



Note
Currently, only Microsoft Visual Studio 2008 and Visual C__ Express 2008 are supported for the features described here.

Start IDE:

1. Checked by default.
2. If checked, after Code Generation (e.g. *C++ Code Generation* (users), or *IBIS AMI Code Generation* (users)) is complete, IDE will be started with the Code Generated content loaded into it.

Automatically compile after code generation:

1. Unchecked by default.
2. If checked, the Code Generation generated content will be automatically built into targeted library.

Load SystemVue library after compilation:

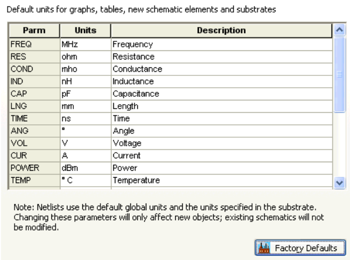
1. Only active if **Automatically compile after code generation** is checked.
2. Only applies to Code Generation targeted at **SystemVue Model Shell** (Refer to *C++ Code Generation* (users) for the choices of targeted shells).
3. Checked by default.
4. If checked, once SystemVue Model DLL is automatically compiled, it will also be loaded into SystemVue immediately for use.

Default Units Options Tab

Use the Global Options Units window to make global changes to the default units in a schematic. Changing the default units has no bearing on any of the parts that are in the schematic. Only the initial units of parts placed after the default unit changes are affected.

The global default units used are listed in the table below.

Quantity	Units
Angle	Degrees
Capacitance	pF (picofarads)
Conductance	mhos (1/ohms or Siemens)
Current	Amps
Frequency	MHz (Megahertz)
Inductance	nH (nanohenries)
Physical Length, Width, Height	mm (millimeters), or based on substrate for netlist
Power	dBm (referenced to a milliwatt)
Resistance	ohms
Temperature	C (Celsius)
Time	ns (nanoseconds)
Voltage	V (volts)

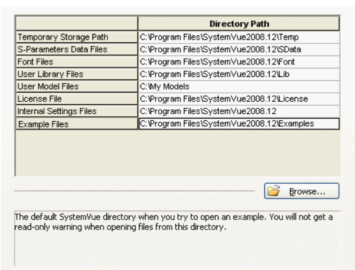


To change the global default units:

1. Click **Tools** on the menu and select **Options**.
2. Click the **Units** tab.
3. Change the units you want by clicking the **Units** grid cell next the parameter type and selecting the desired unit from the pop-up combo box.
4. Click **OK**.

Directories Options Tab

Use the global options directories window to set the default directory paths.



To set the global directory paths:

1. Click **Tools** on the menu and Select **Options.**
2. Click the **Directories** tab.
3. Click on **Directory Path** or label to see a description of what it's used for.

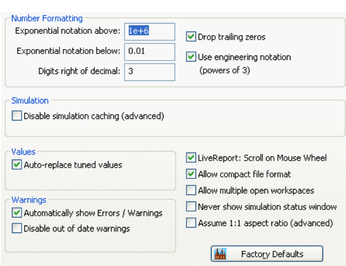
To change a path:

1. Click a **Directory Path**
2. Click **Browse**
3. Select the correct path
4. Click **OK**

> **Note**
> You can edit the path directly.

General Options Tab

Use the Global Options General window to select general environment options not specific to any one area of the program.

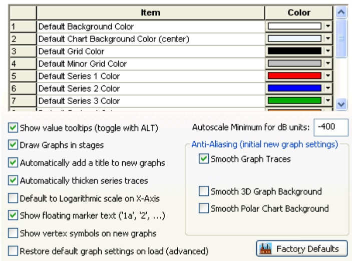


To change general global options:

1. Click **Tools** on the menu and select **Options**.
2. Click the **General** tab.
3. Adjust the settings:
   - **Number Formatting** – Specifies how the program should display numbers. This format is used uniformly throughout (tables, graph axes, dataset displays).
   - **Simulation** – These settings control the simulation engines.
     - Disable simulation caching: Turns off caching of simulation data (runs slower when disabled).
   - **Values** – These settings control parameter values
     - Auto-replace tuned values keeps the tuned values up-to-date.
   - **Warnings** – These settings control the display of Errors / Warnings
     - Automatically show: Instructs the program to show the Errors window when there are errors in the workspace and to hide the window when there are no errors remaining.
     - Disable out of date warnings turns off those warnings.
   - **LiveReport: Scroll on Mouse Wheel** – Option to control the mouse wheel behavior on a Live Report. "Ctrl-Mouse Wheel" will zoom and "Shift-Mouse Wheel" pans right or left. If this is not checked, it will zoom on scroll wheel.
   - **Allow compact file format** – Allows you to save compressed data files.
   - **Allow multiple open workspaces** – Allows more that one workspace (at a time) to be open, so that items may be easily copied from one workspace to another.
   - **Never show simulation status window** – Never show simulation status window during simulations or evaluations.
   - **Assume 1:1 aspect ratio** – Ignores incorrect video device information and assumes that the video display has square pixels. Enable this setting if Smith charts are oval, instead of circular.
   - **Factory Defaults** – When clicked, this button resets all of the settings on this page of the dialog box.
4. Click **OK**.

Graph Options Tab

Use the Global Options Graph window to set global (shared) options for graphs.

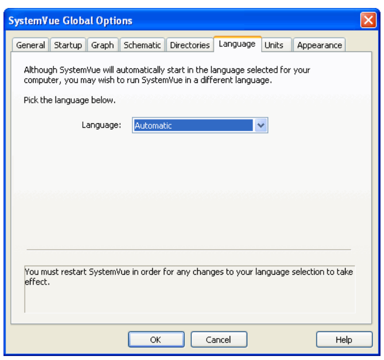


To change the graph global options:

1. Click **Tools** on the menu and select **Options**.
2. Click the **Graph** tab.
3. Adjust the settings:
    - **Item colors** – These colors are used whenever a new graph or series is created. To apply these colors to an existing graph, right-click inside the graph window and select "Set All Colors To Defaults".
    - **Show value tooltips** – Shows the data value in a tool tip window when the cursor is placed over a trace data point.
    - **Draw Graphs in stages** – graphs can draw in stages. A simple graph is drawn first and details are progressively added. This will help with optimizations and sweeps where graphs redraw over and over.
    - **Automatically add a title** – Places a simple title at the top of each new graph.
    - **Automatically thicken series traces** – This setting will widen the lines used to draw the series (trace) line, when a graph is fairly large.
    - **Default to Logarithmic scale** – Switches the X-axis from linear to logarithmic scale.
    - **Show floating marker text** – Enables short marker labels of the form '1a' or '2'. If not checked, no floating marker text will be shown, when graph markers are drawn in the margin on the right.
    - **Show vertex symbols** – Marks series trace vertices with a dot or other symbol (to help distinguish traces on a black & white printout).
    - **Restore default graph settings** – This option is rarely used, but can recover a graph from a damaged workspace file.
    - **Autoscale Minumum** – The lower auto-scale boundry (prevents scaling all the way down to -600dB).
    - **Anti-Aliasing** – These check-boxes enable a smoothing effect to be used when drawing graphs. This gets rid of the stair-stepped, jagged edges when graphs are drawn. When enabled, the graph is drawn with a slightly fuzzy look, which is actually sub-pixel accurate and can accentuate the slight ripples in a trace.
    - **Factory Defaults** – When clicked, this button resets all of the settings on this page of the dialog box.
4. Click **OK**.

Language Options Tab

Use the Global Options Language window to select a different language in which to run. The default language is pre-selected for your computer and is listed as Automatic in this window. Other choices include Chinese, Korean, Russian, and Japanese. You must restart your computer before any language changes can take effect.

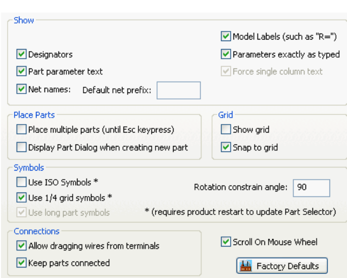


To change the language global options:

1. Click **Tools** on the menu and select **Options**.
2. Click the **Language** tab.
3. Select a language from the Language list.
4. Click **OK**.

Schematic Options Tab

Use the Global Options Schematic window to set options for all schematics.



To change the schematic global options:

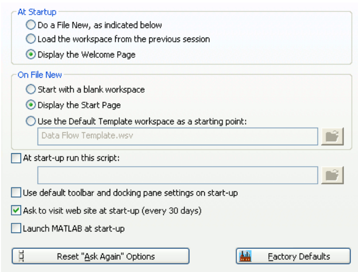
1. Click **Tools** on the menu and select **Options**.
2. Click the **Schematic** tab.
3. Adjust the settings:
    - **Show** – , Designators, Part Parameter Text, etc. Check to enable the specified information to be displayed on a schematic.
    - **Place Parts** – Multiple parts allows parts to be placed each time you left-click on the schematic. Press the Esc key to stop dropping parts. Display Part Dialog will bring up the part dialog each time a part is placed on a schematic, so that the parameters may be entered.
    - **Grid** – Show the background grid and snap the mouse cursor to the grid (if enabled).
    - **Symbols** – Use ISO symbols: When checked, ISO standard symbols will be

placed. (The ISO standard resistor is a box, instead of a zigzag.) Use ¼ grid symbols: When checked, it will place ADS-compatible parts that have terminals spaced on ¼ of the standard part length (which i the length of a resistor). standard parts are on a 1/6th grid spacing. These settings will not take full effect until you have exited and restarted. Rotation constrain angle: Sets the F3-key rotation increment (usually 45 or 90 degrees).
- **Connections** – Allow dragging wires enables schematic parts to be easily connected; just place the mouse cursor over a part terminal, press the left-button, and drag the newly-created connector to another node. Keep parts connected ensures that schematic parts retain their electrical connections, by inserting new wires (as necessary) when dragging parts. The Alt-key acts as a toggle for the keep connect setting.
- **Scroll On Mouse Wheel** – When checked, the mouse center wheel scrolls the schematic window; when unchecked, the wheel zooms the window instead.
- **Factory Defaults** – When clicked, this button resets all of the settings on this page of the dialog box.

4. Click **OK**.

Startup Options Tab

Use the Global Options Startup window to customize start up.



To change the startup global options:

1. Click **Tools** on the menu and select **Options**.
2. Click the **Startup** tab.
3. Adjust the settings:
   - **At Startup** – Specifies the action taken each time the program is run.
   - **On File New** – Specifies the action taken whenever a File / New action is initiated.
   - **At startup run this script** – Allows a custom startup action.
   - **Ask to visit web site at start-up** – Will cause a dialog box to be shown every 30 days asking if the user wants to check the web for updates.
   - **Use default toolbar settings on startup** – Forces the program to reinitialize the toolbars at startup.
   - **Factory Defaults** – When clicked, this button resets all of the settings on this page of the dialog box.
4. Click **OK**.

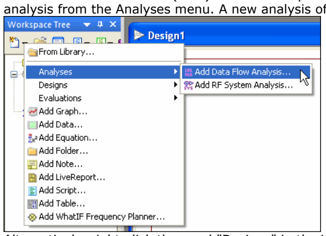
Analysis

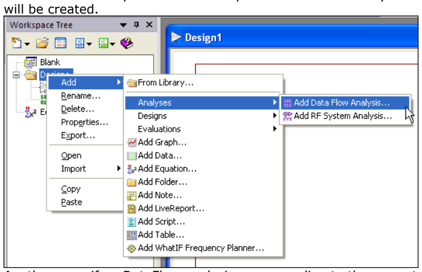
Circuits and systems can be analyzed in many different ways. When you simulate a circuit, the settings for the analysis determine how the simulation runs. The analysis creates a dataset with the simulation results. If an analysis is set to *automatically recalculate* it will re-simulate each time you make a change to the schematic design and then click a graph or table dependent on the analysis.

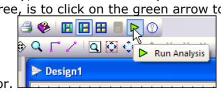
SystemVue provides the following analysis engines:

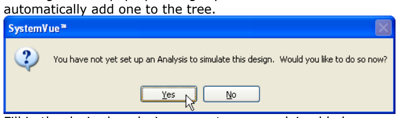
- *Data Flow* (sim) - Performs a data driven analysis on data driven models.
- *RF Design Kit Spectrasys* (sim) - Performs a system-block-level non-linear analysis on the entire system to determine if all system-level requirements are met.

To add an analysis

1. Click the New Item button ( ) on the Workspace Tree toolbar and select an analysis from the Analyses menu. A new analysis of the selected type will be created.


2. Alternatively, right-click the word "Designs" in the Workspace Tree, select "Add", and then select an analysis from the Analyses menu. A new analysis of the selected type will be created.


3. Another way, if no DataFlow analysis corresponding to the current schematic exists on the tree, is to click on the green arrow toolbar button to the right of the calculator.


A dialog box will pop up asking if you'd like to create one and if you click yes, it will automatically add one to the tree.


4. Fill in the desired analysis parameters as explained below.
5. When you click OK or Calculate the analysis will run and create a data set.

Annotations

Annotations include text boxes, arrows, shapes, and controls (widgets) that can be placed on a schematic, graph, or LiveReport to help document a workspace, highlight items of interest, etc.

Tools	Purpose
Rectangle	Draw a square or rectangle.
Ellipse	Draw a circle or ellipse
Polygon	Draws a filled polygon or unfilled polyline.
Arrow/Line	Draw a line or arrow. Change the arrow style by selecting a line and picking an arrow type from Arrows button menu.
Arc	Draw a circular arc.
Picture	Insert a picture. Use this annotation to add a company logo to a graph, for example. Double-click the new object and select a JPG, GIF, or BMP image file to be displayed. (To allow all users to see the image, the bitmap file should reside on a network server.)
Text	Place text. Text has a number of settings. Double-click a text annotation to set the horizontal and vertical justification (text alignment). The name of the text item can be changed and shown on-screen, which simplifies building a schematic title block.
Text Balloon	Draw a text balloon. This annotation has a "tail" which can be anchored to a data point on a graph, to the page, or not anchored (using the right-button menu).
Button	Draw a user button. This annotation can be "clicked" to run a custom script, which is specified by double-clicking the outer EDGE of the button control. (The middle of the button runs the script.)
Slider	Draw a slider control. This annotation is linked to a tunable parameter and functions much like the *Tuning Window* (users).
Settings	**Purpose**
Fill Color	Sets the annotation fill color. Use the 3 color buttons to change the colors of the selected annotation(s). New annotations will be created using the current colors. The bottom-right color swatch (with a diagonal slash) is transparent, which specifies an unfilled object.
Line Color	Sets the annotation line / border color. The bottom-right color swatch (with a diagonal slash) is transparent, which specifies a object with no outline.
Text Color	Sets the annotation text color.
Line Thickness	Set the width of borders and lines.
Line Style	Set the drawing style of borders and lines (dash pattern, etc.).
Arrows	Set the arrow style of lines.
Properties	Display the properties window for the selected annotation.

Contents

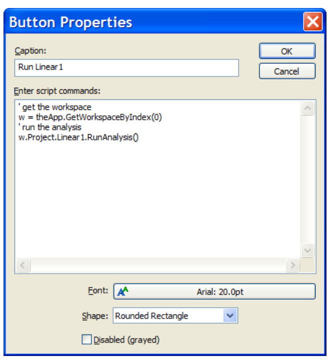
- *Creating Annotations* (users)
- *Line Annotations* (users)
- *Text Annotations* (users)
- *Button Annotations* (users)
- *Slider Annotations* (users)
- *Variable Selector* (users)

Button Annotations (Widgets)

A button annotation is a control which runs a script when clicked. Buttons and other widgets are initially created using "stock Windows colors"; the controls' colors can easily be changed using the Annotation toolbar, as can line thickness, etc.

To change the properties of a button:

1. Double-click the *EDGE* of any button object.
2. Make the changes you want.
3. Click **OK**.



Property	Purpose
Caption	The title text displayed on the button.
Script Commands	Specifies the script to be run, when the button is clicked.
Font	To set the font.
Shape	Buttons can be a rectangle, a rounded rectangle, or an ellipse.
Disabled	Grays and inactivates the button.

Creating Annotations

The Annotation button  on the Schematic, LiveReport, and Graph toolbars toggles the display of the Annotation toolbar.

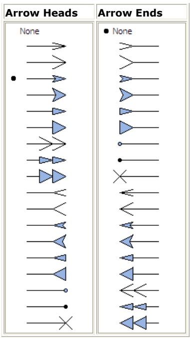
The toolbar provides tools like lines, circles, and text that you can use to point out details of interest on a schematic, draw a box around a group of components, etc.

To place an annotation:

1. Click the various settings buttons (colors, line style, etc.) to adjust the settings for newly created annotations
2. Click an annotation tool button (box, arc, text, etc.) on the Annotation Toolbar.
3. Click in a schematic, LiveReport, or graph window to place the new annotation.
4. Use the annotation setting buttons to change existing, selected annotations. (More than 1 annotation can be adjusted at a time).
5. To set the Font for annotations with text, right-click the object and pick **Font...** from the pop-up menu.

Line Annotations

Lines have many drawing options: Line Thickness, style, color, arrowheads, etc., which are controlled via the Annotation toolbar and by the object's right button menu. Lines can have arrowheads and ends. Simply select a line and pick an arrow type:

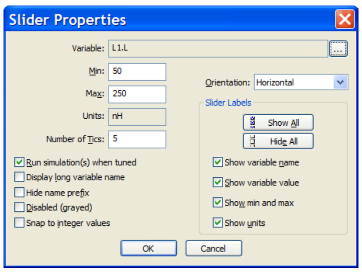


Slider Annotations (Widgets)

A slider annotation is a control which adjusts a tunable equation variable, part parameter, etc. Sliders and other widgets are initially created using "stock Windows colors"; the controls' colors can easily be changed using the Annotation toolbar, as can line thickness, etc.

To change the properties of a slider:

1. Double-click the EDGE of any slider object.
2. Make the changes you want.
3. Click **OK**.



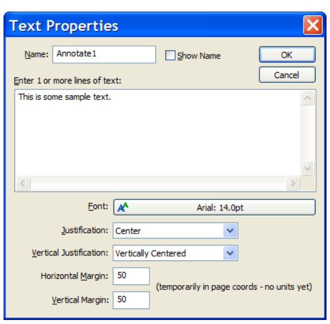
Property	Purpose
Variable	The variable to be tuned.
'...' Button	Brings up a selector to pick the variable.
Min and Max	Specifies the limits of the tuning range. These can be set to the names of equation variables, for adjustable limits.
Units	Displays the units of the tune variable.
Number of Tics	The number of slider division marks.
Orientation	Sliders can be horizontal or vertical.
Slider Labels	Specifies what text (if any) to display on a slider.
Show / Hide All	Check or uncheck all the Show checkboxes.
Run simulations	Runs enabled simulations on left-button up.
Display long variable name	Displays the tune variable's long name (Eg: Project\Sch1\L1.L).
Hide name prefix	Omits the part or equtaion name (Eg: L instead of L1.L).
Disabled	Grays and inactivates the slider.
Snap to integer values	Limits the tuning to integer values.

Text Annotations

A text annotation is a filled rectangular box with text inside.

To change the properties of a text annotation:

1. Double-click any text object.
2. Make the changes you want.
3. Click **OK**.



Property	Purpose
Name	The name of the Text object.
Show Name	Displays the name of the text item, which simplifies building a adding a title block or other "labeled text".
Enter Lines of Text	Specifies the test to be displayed.
Font	Click the button to set the font.
Justification	Sets the horizontal justification (alignment) of the text: Left, Right, or Center.
Vertical Justification	Sets the vertical alignment of the text: Top, Bottom, or Vertically Centered.
Horizontal and Vertical Margins	Sets the margins (border gap) of the text. Specified in page coordinates (1/1000ths of an inch).

Tips for advanced users

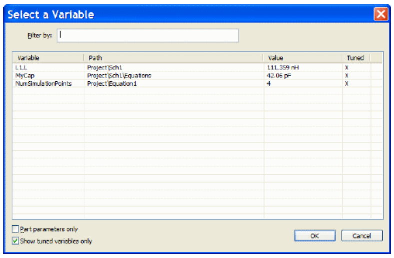
Text annotations can use equations. For example, if your workspace contains an equation block with a text variable named CompanyName, you can place =CompanyName in the Text field. (The leading = sign indicates that the text string is actually an expression.) When the annotation is drawn, the equation will be evaluated and the result displayed.

Text annotations can display model and parameter info when used *within a* ***custom symbol*** . This is implemented via macro-text-substitution. When symbol text is drawn on a schematic, the displayed text is modified prior to output. For example, Name=%Model% would be displayed as "Name=Resistor" on a symbol using a resistor model. The recognized macro strings are:

1. **%Des%** - Displays the part's designator.
2. **%Model%** - Displays the name of the model attached to the part.
3. **%MODEL%** - Displays the model name in UPPERCASE.
4. **%ParameterName%** - Displays the value of the specified model parameter attached to the part. E.g. R, C, L, QL, MODE, etc.

Variable Selector

This is displayed via the **'...'** button.



Property	Purpose
Filter by	Limits the variables displayed to only those that include the specified text.
Variable	Displays the variable's name.
Path	Displays the full pathname of the variable.
Value	Displays the current value of the variable.
Tuned	Displays an X, if the variable is tunable.
Part parameters only	Limits the variables displayed to only part parameters.
Show tuned variables only	Limits the variables displayed to only tunable variables.

C++ Code Generation

SystemVue **C++ Code Generator** allows users to generate C++ code for a network (or a sub-network) of a system. The generated code is in SystemVue *C++ Model* (users) format but can be wrapped in different targets (e.g. ADS Ptolemy Model) and used in other simulation environments. The generated *C++ Model* (users) contains the following contents to implement the system of the code generation network:

- Declaration of models and sub-networks inside the network.
- Declaration of interface ports and specifying the interface data flow rates of the network.
- Specifying model parameters.
- Allocation (and de-allocation) of buffer memories for transferring data inside the network.
- Setting up *models' input and output ports* (users) for reading and writing data from and to *circular buffers* (users).
- Executing pre-scheduled model executions for a complete *data flow schedule iteration* (sim) of the network.

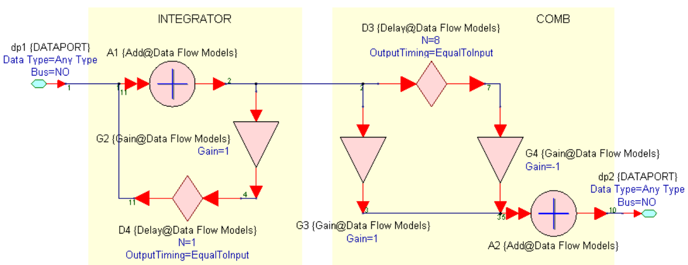
The generated C++ Model is also generic enough to be used as standalone C++ code for other applications.

Quick Start

In this section code generation flow is used to generate C++ code for a CIC filter. The similar flow can be used for more complex systems build with code-generation supported models in SystemVue. All user defined *C++ models* (users) following the directions in the section Writing C++ Models for Code Generation support code-generation.

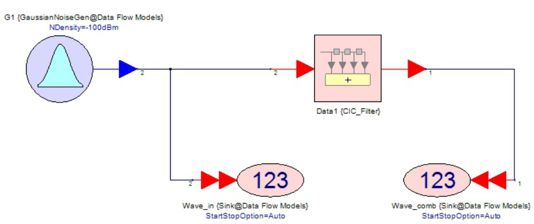
Creating a Sub-network Model

Create a sub-network model implementing a CIC filter using models in Algorithm Design library as follows,



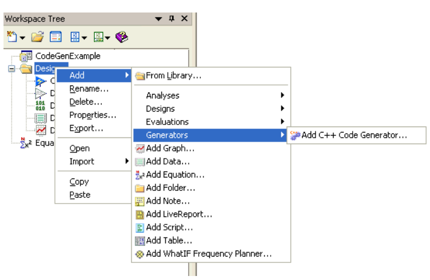
Creating a Design using the Sub-network Model

Create a Design using the CIC filter sub-network model as follows

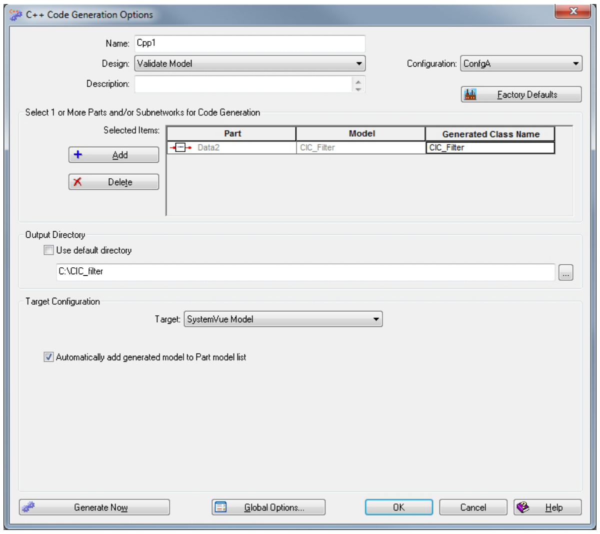


Adding a C++ Code Generator Analysis

Right click on the workspace tree and add a C++ Code Generator Analysis as follows

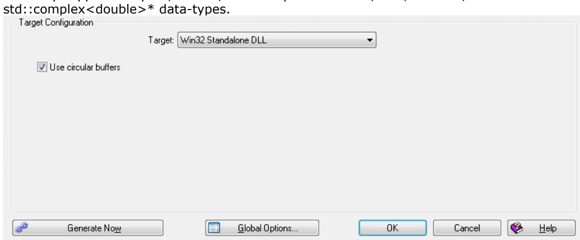


In the **C++ Code Generation Options** dialog box, edit the Name to CIC_CodeGeneration and select Top Level Design to be Design1. The dialog should look as shown below:

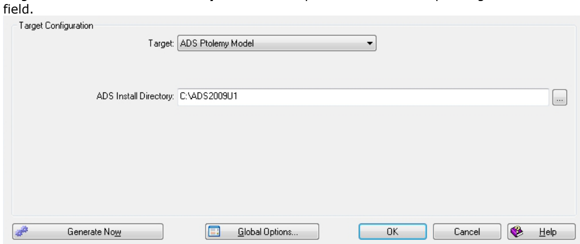


The different fields and buttons of the dialog are explained below:

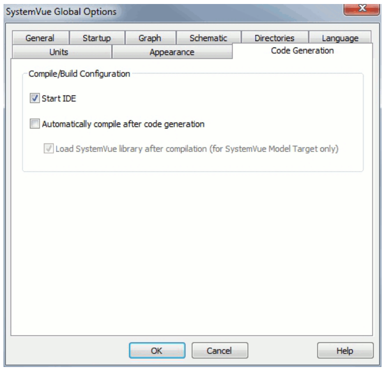
- The **Name** is the C++ Code Generator name, which identifies it on the workspace tree.
- The **Design** selects the top level design for code generation.
- The **Configuration** selects the Model Manager configuration of the **Design** selected. The default configuration is "**Default ...**" which will generate the code for the design as it appears in the schematic (for most cases, this is the value you need to select). It is an advanced option, please read *Modifying a Design* (users) section for more details.
- The **Add** button selects one or more parts for code generation; the generated code will be for the models associated with the selected parts.
- The **Delete** button removes parts that have been previously added.
- The **Selected items** grid lists the parts for which code will be generated.
  - **Part** is the full path to the part selected for code generation; this is a non-editable field.
  - **Model** is the name of the model the part was using when selected for code generation; this is a non-editable field.
  - **Generated Class Name** is the C++ class name that will be used when generating code for the corresponding part. This is editable and can be modified if the default auto-generated name is not desired.
  - You can add multiple parts for code generation at a time.
- The **Output Directory** is the directory where the Visual Studio Solution will be generated.
- If **Use default directory** is selected then the default directory (a directory with the same name as the workspace located in the same directory as the workspace) is used.
- The **Target** drop down menu specifies the target (application) on which the generated code will run. The available targets are: SystemVue Model, Win32 Standalone DLL, ADS Ptolemy Model. See Supported Targets section for more details.
  - **SystemVue Model** will generate code for a SystemVue Model that can then be imported and run inside SystemVue.
  - **Win32 Standalone DLL** will generate code that can be compiled in a standalone dll for use in other applications. For this target the **Use circular buffers** checkbox controls whether the generated C++ model is going to use **CircularBuffer** or **GenericType** input/output interface. Generic-type interface currently supports only int, double, std::complex<double>, int*, double*, and std::complex<double>* data-types.



  - **ADS Ptolemy Model** will generate code and an associated pl file(s) that can be compiled in a dll for use in the ADS Ptolemy simulation environment. For this target the **ADS Install Directory** needs to be specified in the corresponding field.

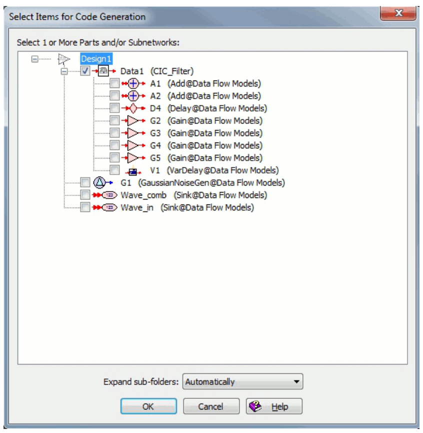


- The **Global Options** opens the global options for Code Generation as shown below. The same dialog can be opened using Tools->Options and then selecting Code Generation tab.



- Selecting **Start IDE** opens the Visual Studio IDE after code-generation.
- Selecting **Automatically compile after code generation** compiles corresponding generated visual studio project after code generation.
- Selecting **Load SystemVue library after compilation** loads the dll after compilation in SystemVue. This option is valid only for SystemVue Model target only.
  As clear from its name, any option set under Global Options are set for all future use of any code generator and not only for the instance of Code Generator used to invoke global options.

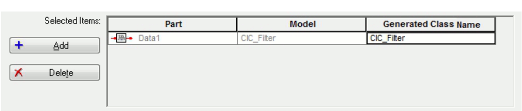
Clicking on the Add button brings up a dialog box where the sub-networks for which C++ code generation is desired can be selected. For this example, select "Data1 (CIC_filter)" for code generation as follows



Click **Expand sub-folders** (if desired). (Note that you can open any individual sub-folder by clicking the + symbol on its left.)

- **Automatically** opens sub-folders with only a small number of parts.
- **Always** opens all sub-folders.
- **Never** only opens the Top Level Design folder, but leaves all the sub-folders closed.

Click **Ok** button. The **Selected items** grid in C++ Code Generation Options dialog box should look like as follows.



Generating Code

Clicking on the **Generate Now** button (or right-clicking on the C++ Code Generation item on the workspace tree and selecting **Generate Now**) generates the C++ code plus other necessary files (e.g. pl files for the **ADS Ptolemy Model** target) as well as an associated Visual Studio solution and project(s) that can be used to build the code. The Visual Studio project for the different targets (SystemVue Model, Win32 Standalone DLL, ADS Ptolemy Model) is created under a different directory (SystemVue, StandaloneDLL, Ptolemy) in the top level Visual Studio solution directory. All Visual Studio projects created from the same C++ Code Generator are included in the same Visual Studio solution. The first time the Visual Studio solution is created, Visual Studio is launched, the generated solution and project files are loaded, and Visual Studio comes to the foreground. All that needs to be done after that is selecting the desired configuration (Release/Debug) and building the solution. Every time the Visual Studio solution is updated (e.g. new project added to the solution, more files added to an existing project in the solution) through the code generator, Visual Studio first saves the solution (this guarantees that changes the user has made are not overwritten), then closes the solution, then the code generator updates the necessary files, and finally Visual Studio loads the updated solution.

> When selecting a C__ model part (non-subnetwork part) for code generation, SystemVue will generate a C__ wrapper inherited from the original C__ model. For this reason, the **Generated Class Name** should be different than the full class name (including namespace) of the original C__ model. In general, users just need to take care of user-defined C__ models as SystemVue built-in models are protected within a namespace.

Supported Targets

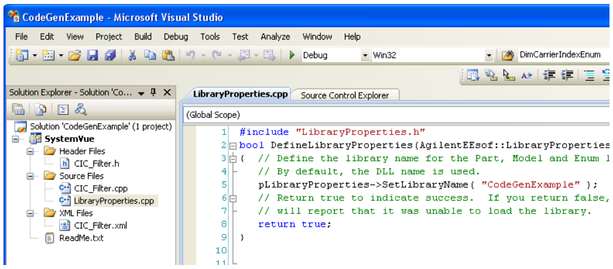
This section describes in more detail the supported target types. Although the generated files are very similar or identical for all target types the compiler and linker options used to build the code are different.

SystemVue Model

The SystemVue Model target generates code that can be built into a DLL for use inside SystemVue. The DLL can be loaded into SystemVue using the *Library Manager* (users) (see section *Adding C++ Custom Libraries* (users)). The generated library can also be loaded automatically using "Global Options..." button in "C++ Code Generation Options" dialog by selecting "Automatically compile after code generation" and "Load SystemVue library after compilation" options.

If the option "Automatically add generated model to Part model list" is selected for SystemVue Model target, then the generated model will be added in the managed model list of corresponding part. The instance name will be the Name of code generator, and the library name is the name of auto-generated library name which is the name of the output directory by default. If you change the library properties manually after generating the code then you will have to update the managed model list for corresponding models manually.

The structure of the auto-generated Visual Studio solution and project is shown below.

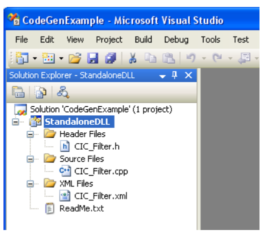


The solution name is the same as the workspace name (CodeGenExample) and the project name is SystemVue.

- The *Header Files* folder contains the header file(s) for the generated classes.
- The *Source Files* folder contains the implementation (.cpp) file(s) for the generated classes. In addition, the *Source Files* folder contains the file *LibraryProperties.cpp*, which can be used to change the name of the Part, Model, and Enum libraries created when the DLL is loaded into SystemVue. By default the name of these libraries is the workspace name.
- The *XML Files* folder contains xml file(s) that describe the model interface, that is, the names, types, and other properties of the the model's parameters, inputs, outputs, etc. These xml files are not necessary for building the DLL.

Win32 Standalone DLL

The Win32 Standalone DLL target generates code can be built into a DLL for use outside of SystemVue. The structure of the auto-generated Visual Studio solution and project is shown below.



The solution name is the same as the workspace name (CodeGenExample) and the project name is StandaloneDLL.

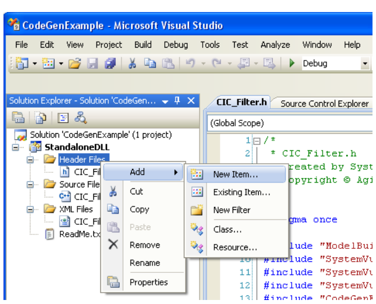
- The *Header Files* folder contains the header file(s) for the generated classes.
- The *Source Files* folder contains the implementation (.cpp) file(s) for the generated classes.
- The *XML Files* folder contains xml file(s) that describe the model interface, that is, the names, types, and other properties of the the model's parameters, inputs, outputs, etc. These xml files are not necessary for building the DLL.

Exporting symbols from the Standalone DLL

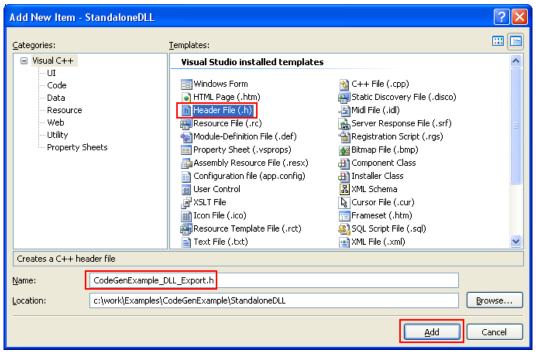
No symbols are exported from the DLL that is built. Symbols are required to be exported if you wish to reference the functions and classes defined in this DLL from another DLL or an executable. To export symbols from a DLL, you need to use:

```
__declspec(dllexport)
```

To do this, add a new header file to the StandaloneDLL project. A good practice is to call this header file "<Solution Name>_DLL_Export.h". To add a new file right click on the *Header Files* folder and select **Add > New Item ...**

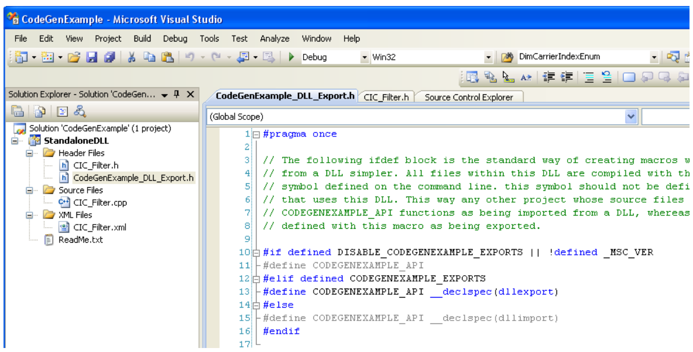


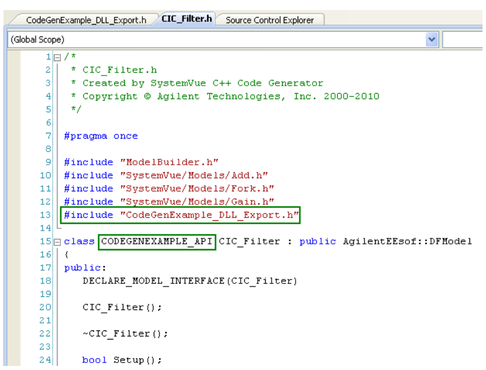
In the dialog that pops up select Header File (.h), type the name of the new file in the Name field, and press the Add button.



An empty file called CodeGenExample_DLL_Export.h is created and opened in Visual Studio for editing. Copy the content shown below and paste it in this file. Replace CODEGENEXAMPLE with the name of your solution.

```
#pragma once
// The following ifdef block is the standard way of creating macros which make exporting
// from a DLL simpler. All files within this DLL are compiled with the CODEGENEXAMPLE_EXPORTS
// symbol defined on the command line. this symbol should not be defined on any project
// that uses this DLL. This way any other project whose source files include this file see
// CODEGENEXAMPLE_API functions as being imported from a DLL, whereas this DLL sees symbols
// defined with this macro as being exported.
#if defined DISABLE_CODEGENEXAMPLE_EXPORTS || !defined _MSC_VER
#define CODEGENEXAMPLE_API
#elif defined CODEGENEXAMPLE_EXPORTS
#define CODEGENEXAMPLE_API __declspec(dllexport)
#else
#define CODEGENEXAMPLE_API __declspec(dllimport)
#endif
```





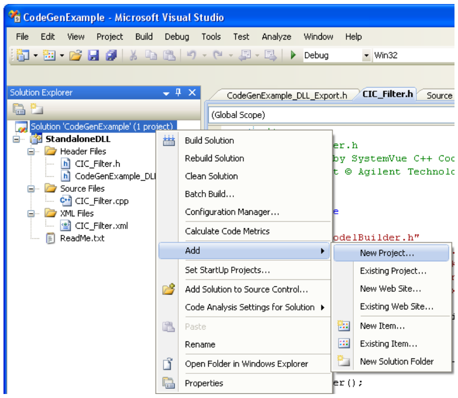
Now, you can modify the generated code to export the classes and functions you wish to reference from other DLLs or executables. For example, to export the generated CIC_Filter class, modify "CIC_Filter.h" by adding an include statement for the header file you just added and adding the API preprocessor definition (CODEGENEXAMPLE_API) to the declaration of the class.

> ⚠ If you generate code from SystemVue again, the file CIC_Filter.h will be regenerated and the edits you have made above will be lost. In this case, a window with a warning that certain files will be overwritten pops up and you can choose to overwrite the files or not.

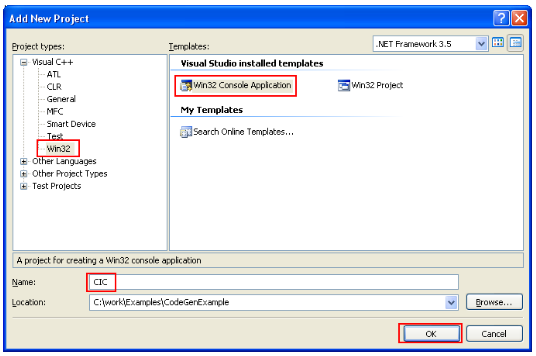
Now the CIC_Filter class/model can be instantiated and used in other applications. An example on how to use a model from a standalone DLL in a standalone executable is described in the next section.

Using a model defined in a Standalone DLL in an executable

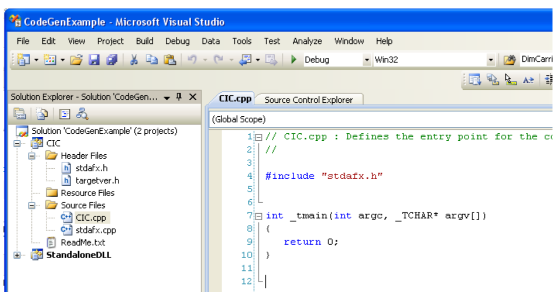
1. Make sure you have exported the classes/models you want to use in your executable (see previous section).
2. Add a new project to the solution. Right click on the solution and select **Add > New Project ...**

2. 

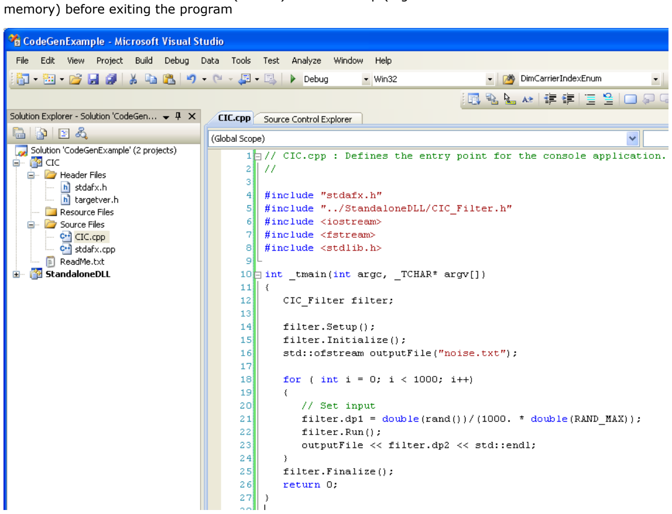
3. In the dialog that pops up select **Win32** under **Visual C++** in the *Project types* area, then select **Win32 Console Application** in the *Templates* area, type the name of the new project in the Name field, and press the OK button.



4. In the new dialog that pops up just press the **Next** and then the **Finish** button.
5. The new project is created and added to the solution. The CIC.cpp file, which contains the _tmain function, is automatically opened for editing.



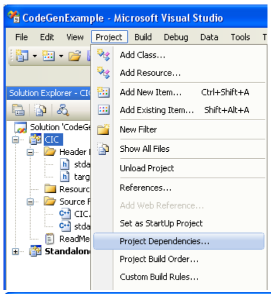
6. Edit this file to implement the application you want. The code shown below (this code is also provide as a code snippet below so that you can copy/paste it and try it out yourself)
   - instantiates the CIC_Filter class (line 12) that was generated by SystemVue
   - initializes it (lines 14 and 15)
   - passes random data to it (line 21)
   - runs it (line 22)
   - writes the filtered output to a file (line 23)
   - calls the finalize method of the filter (line 25) to do clean up (e.g. free allocated memory) before exiting the program

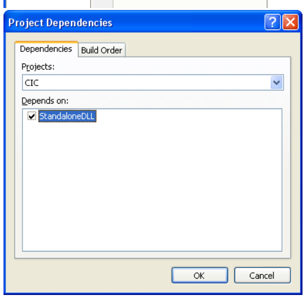



```
// CIC.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include "../StandaloneDLL/CIC_Filter.h"
```

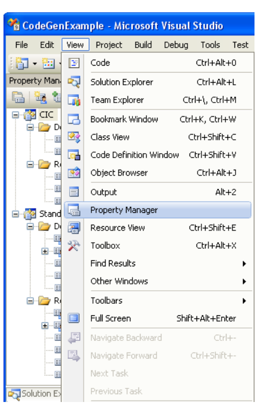
```
#include <iostream>
#include <fstream>
#include <stdlib.h>
int _tmain(int argc, _TCHAR* argv[])
{
 CIC_Filter filter;
 filter.Setup();
 filter.Initialize();
 std::ofstream outputFile("noise.txt");
 for ( int i = 0; i < 1000; i++)
 {
 // Set input
 filter.dp1 = double(rand())/(1000. * double(RAND_MAX));
 filter.Run();
 outputFile << filter.dp2 << std::endl;
 }
 filter.Finalize();
 return 0;
}
```

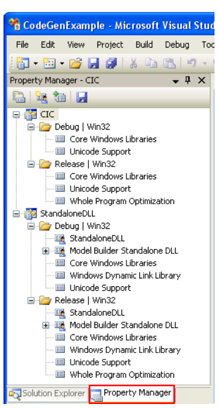
7. Add a dependency between the CIC and the StandaloneDLL projects.
   - From the *Project* menu select **Project Dependencies...**.
   - In the Project Dependencies dialog that pops up, go to the Dependencies tab, select CIC in the Projects drop down menu, check the checkbox next to StandaloneDLL in the Depends on area, and press the OK button.



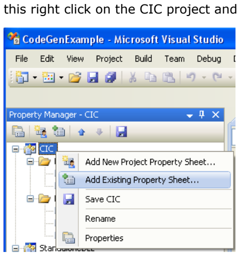


8. Add the appropriate Property Sheets.
   - From the *View* menu select **Property Manager**. The Property Manager tab appears net to the Solution Explorer tab.

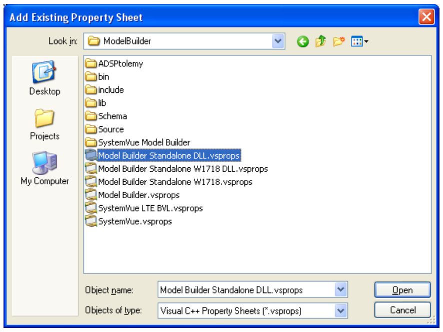




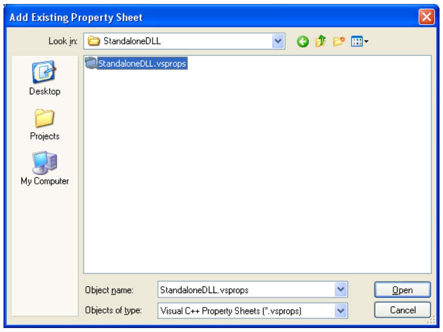
   - Add the "Model Builder Standalone DLL" property sheet to the CIC project. To do this right click on the CIC project and select **Add Existing Property Sheet...**.



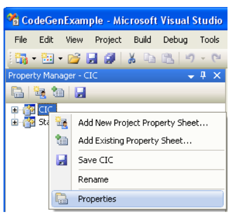
   Navigate to the Modelbuilder directory of your SystemVue installation, select "Model Builder Standalone DLL.vsprops", and press the open button.



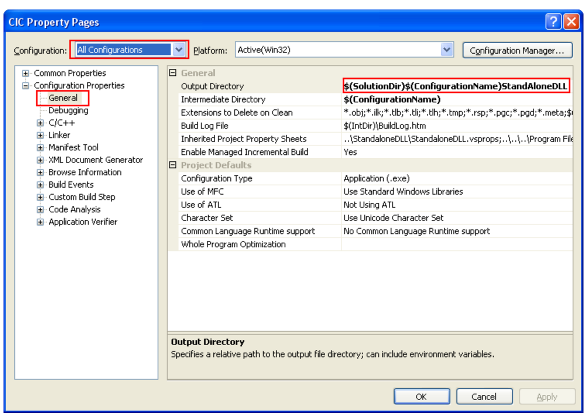
- Repeat the above step to add the "StandaloneDLL" property sheet to the CIC project. This property sheet is under the StandaloneDLL directory of your solution directory (c:\work\Examples\CodeGenExample for the example discussed here).



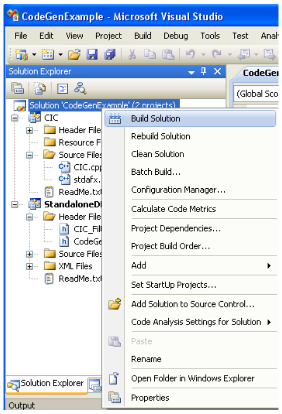
9. Change the Output Directory for the CIC project so that the executable is built in the same directory as the standalone DLL.
   - Right click on the CIC project and select **Properties**.



   - In the dialog that pops up, select **All Configurations** in the *Configuration* drop down menu, go to the **General** section under the *Configuration Properties*, set the Output Directory to "$(SolutionDir)$(ConfigurationName)StandAloneDLL" (the default value is "$(SolutionDir)$(ConfigurationName)"), and press the OK button.



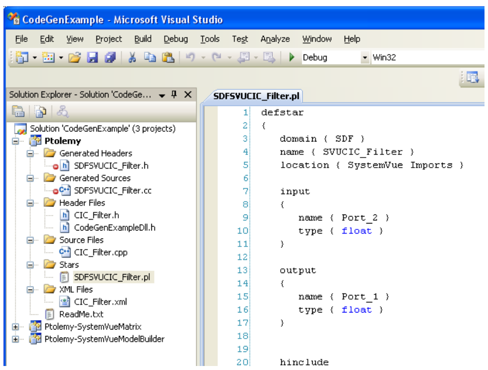
10. Build the solution by right clicking on the solution name and selecting **Build Solution**.

10. 

The CIC.exe executable is created under the DebugStandAloneDLL/ReleaseStandAloneDLL directory of your solution directory (c:\work\Examples\CodeGenExample for the example discussed here) depending on whether you chose to build Debug or Release code. You can navigate to this directory (in Windows Explorer) and run it by double clicking on it. You will see the noise.txt file being created. You can also place breakpoints and debug it inside Visual Studio.

ADS Ptolemy Model

The ADS Ptolemy Model target generates code and all other necessary files that can be built into a DLL for use inside the ADS Ptolemy simulation environment. The structure of the auto-generated Visual Studio solution and project is shown below.

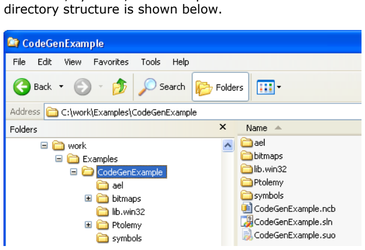


The solution name is the same as the workspace name (CodeGenExample) and the project name is Ptolemy. The solution includes two more projects: *Ptolemy-SystemVueMatrix* and *Ptolemy-SystemVueModelbuilder*. These projects are not copied into the solution directory. They exist under the directory ***My Documents\SystemVue\<SystemVue version>\<ADS version>*** (without the Ptolemy- prefix) and they are included in the solution from the above directory. These projects are built as part of the solution and the DLLs they create are needed so that the DLL built from the Ptolemy project works properly. Do not make any changes to the files of these two projects. The structure of the Ptolemy projects is described below:

- The *Header Files* folder contains the header file(s) for the generated classes. In addition, the *Header Files* folder contains the file <WorkspaceName>Dll.h., where <WorkspaceName> is the name of the workspace, which is required for compiling. Do not delete or modify this file.
- The *Source Files* folder contains the implementation (.cpp) file(s) for the generated classes.
- The *Stars* folder contains the Ptolemy Language (.pl) file(s), which wrap the generated classes with an ADS Ptolemy model.
- The *XML Files* folder contains xml file(s) that describe the model interface, that is, the names, types, and other properties of the the model's parameters, inputs, outputs, etc. These xml files are not necessary for building the DLL.
- The *Generated Header Files* folder contains the header file(s) for the ADS Ptolemy model(s).
- The *Generated Sources* folder contains the implementation (.cc) file(s) for the ADS Ptolemy model(s).

The header files under the *Generated Header Files* folder and the .cc files under the *Generated Sources* folder are auto-generated from the pl files and do not exist during the creation of the project. They are generated the first time the project is built. Do not delete of modify these files.

When the Ptolemy project is built it creates a DLL (under the lib.win32 directory) as well as the ael, symbols, and bitmaps needed for the model to be used in ADS. The resultant directory structure is shown below.



To use these models in ADS just set the *ADSPTOLEMY_MODEL_PATH* environment variable to the directory where the lib.win32, ael, symbols, bitmaps directories are located (for the example shown above *ADSPTOLEMY_MODEL_PATH* should be set to c:\work\Examples\CodeGenExample) and start ADS. The models will be located under the *SystemVue Imports* library.

Use of SystemVue matrix models in ADS

If a model with an output of SystemVue matrix is connected to a NumericSink it is required that a Gain_M, GainInt_M, or GainCx_M component with Gain=1 is inserted between the model and the sink.

Supported ADS versions

The ADS Ptolemy Model target is compatible with the following ADS versions: ADS 2009 Update 1, ADS 2010.

Creating ADS Ptolemy models for an entire SystemVue Modelbuilder library

If you have a SystemVue library (dll) of custom models and want to use these models in ADS you can follow the process described here to generate the associated pl files and build a dll with the corresponding Ptolemy models. The alternative is to use the Code Generator, where you add each one of the models in your library. This may not be practical if your library contains a lot of models.

1. First create a simple subnetwork model (you can use the CIC_Filter example under Examples\Model Building) and use the Code Generator to generate code using the **ADS Ptolemy Model** target. This step will create the proper Visual Studio solution and project structure with the correct settings. Once this is done you can actually remove the files created by the Code Generator (CIC_Filter.h, CIC_Filter.cpp, CIC_Filter.xml, SDFSVUCIC_Filter.pl, SDFSVUCIC_Filter.h, and SDFSVUCIC_Filter.cc) from the Ptolemy project. Do not remove the <SolutionName>Dll.h header file under the *Header Files* folder. To simplify the description of the next steps we will assume that:
    - the SystemVue installation is under c:\Program Files\SystemVue2011.03
    - the Visual Studio solution directory is c:\work\Examples\CodeGenMyModels
    - the SystemVue custom model library is MyModels.dll
2. Open a DOS window and go to the Ptolemy directory of your Visual Studio solution
   cd c:\work\Examples\CodeGenMyModels\Ptolemy
3. Run the command (make sure the directory where MyModels.dll is located is in your PATH variable)
   c:\Program Files\SystemVue2011.03\bin\SystemVue.exe -XML MyModels.dll
   This command will create the xml file MyModels.xml, which fully describes the interface of your SystemVue custom models.
4. Run the command
   c:\Program Files\SystemVue2011.03\bin\SystemVueModelShell.exe -list -o c:\work\Examples\CodeGenMyModels\Ptolemy -ptolemy MyModels.xml
   This command will create a pl file wrapper for all the models in the MyModels.dll library. The names of the created pl files are SDFSVU<ModelName>.pl.
5. In the Visual Studio *Solution Explorer* window, right click on the *Stars* folder of the Ptolemy project and select **Add > Existing Item...**. In the dialog that pops up, select all the SDFSVU<ModelName>.pl files and press the Add button. This will add the selected pl files under the *Stars* folder.
6. In the Visual Studio *Solution Explorer* window, select all the pl files under the *Stars* folder (you can do this by left mouse clicking on the first pl file and then holding down the *Shift* key and left mouse clicking on the last pl file), then right click and choose **Compile**. This will generate the corresponding SDFSVU<ModelName>.h and SDFSVU<ModelName>.cc files.
7. Add the .h files under the *Generated Headers* folder and the .cc files under the *Generated Sources* folder by right clicking on the folder and selecting **Add > Existing Item...**.
8. Update the Ptolemy project properties so that the C++ compiler has access to the header files of your SystemVue custom models and the linker has access to the associated lib file (make sure the classes representing your models have been exported properly so that they can be referenced from another dll; you can follow a process similar to the one described in Exporting symbols from the Standalone DLL).
9. Build the solution.

Limitations

When writing fixed point C++ models for SystemVue, users can simply override *AgilentEEsof::DFFixedPointInterface::SetOutputFixedPointParameters()* method to let SystemVue automatically set fixed point parameters (including word length, integer word length, sign bit, etc) for each AgilentEEsof::FixedPoint object in AgilentEEsof::FixedPointCircularBuffer. See *Writing Fixed Point Models* (users) for details. However, such automation process is not available in ADS Ptolemy, so users have to modify the source code. If the output fixed point parameters can be derived from model parameters, users can set the fixed point parameters for each AgilentEEsof::FixedPoint object in AgilentEEsof::FixedPointCircularBuffer in the Initialize() method. If the output fixed point parameters depend on the input fixed point parameters, users can set the output fixed point parameters for each output data in the Run() method.

In addition, because ADS Ptolemy and SystemVue use different fixed point data types, conversions between two data types are performed in the generated Ptolemy Language (.pl) model. The conversion functions are coded in \ModelBuilder\include\SystemVue\ADSPtolemy\FixedPointHelper.h under SystemVue installation directory.

Licensing

Using the generated C++ code requires certain SystemVue licenses. The license features required are based on what is included in the design used to generate C++ code and how the generated code is being used (target type).

SystemVue Model Target

If the generated C++ code is used as SystemVue model inside SystemVue then license requirements will be same as running the original design used to generate C++ code. For example if original design contained LTE models then LTE license will be needed.

ADS Ptolemy Model Target

If the generated code is used inside ADS Ptolemy, then SystemVue Core license will not be required, instead ADS Ptolemy license will be used in place of SystemVue Core license. However, if your design to generate C++ code requires any extra license other then SystemVue core then exactly the same license will be needed to use generated code in ADS Ptolemy. For example, if you have used any LTE license in SystemVue design to generate C++ code then you will need exactly the same license to run the design in ADS Ptolemy as well. To use SystemVue specific LTE (or any other non SystemVue Core license) in ADS Ptolemy, please append the SystemVue license path to ADS license environment variable **AGILEESOFD_LICENSE_FILE** along with original ADS license.

Win32 Standalone DLL Target

To use generated code outside SystemVue and ADS Ptolmey, you will need exactly the same licenses as you need for the SystemVue design used to generate C++ code. The SystemVue Core license will always be pulled.

W1718 License

If you have W1718 license available then the first time you run SystemVue C++ code generation using your user account, the source code and corresponding Visual Studio project for SystemVue core models will be copied to "\SystemVue{version}\W1718" under "My Documents" directory for your user account. You can read / modify the code and use it in anyway you want. If you use the libraries created by W1718 source code, and link with the generated C++ code, then you will not need SystemVue core license to use the generated code outside SystemVue, provided that design to generate C++ code contains ONLY SystemVue core model. If the design contains both core models and LTE then both SystemVue Core license and LTE license will be required to use the code outside SystemVue/ADS.

LTE Specific License Requirements

If you have LTE Baseband Verification License then the first time your will generate C++ code, LTE C++ header files will be copied to "\SystemVue{version}\LTE_8.9" under "My Documents" directory for your user account, where LTE_8.9 represents LTE version 8.9. To build any C++ code generated using LTE models in your design requires these headers to be presented in that directory. You will also require LTE license to run use the generated code. If you have purchased LTE Basebad Exploration library then you will have access to complete LTE source code and you can use it in a similar way as W1718 source code.

Schema

Along with C++ code generation, SystemVue generates XML file that describes the interface of the generated C++ model. The XML format is based on the schema provided in \ModelBuilder\Schema\systemvue_model.xsd under SystemVue installation directory. In the same folder, systemvue_model.pdf and systemvue_model.mht are also provided that describe the schema content.

Writing C++ Models for Code Generation

In general, SystemVue C++ code generator supports any C++ model that is created and loaded based on *Creating a Custom C++ Model Library* (users), including user-defined C++ models. However, in order to successfully compile generated code, additional information needs to be provided in *DEFINE_MODEL_INTERFACE* (users) of C++ models that are going to be used in code generation.

- If the class name (say *classname*) of a C++ model is different than the name of the header file that declares the model, then use **ADD_MODEL_HEADER_FILE(*header_file*)** macro to specify the header file. See \ModelBuilder\include\ModelBuilder.h in SystemVue installation directory for macro definition. In this case, *header_file.h* will be included in the generated code. Otherwise, *classname.h* will be automatically included by default.
- If there are headers necessary for the generated code to use a model, and those headers are not included in the model's header file, then use **ADD_MODEL_HEADER_FILE(header_file)** macro to specify the additional headers to be included in the generated code **and** also the model class header.

  > Once ADD_MODEL_HEADER_FILE(header_file) macro is used, C__ code generator will not generate *classname.h*.

- If a C++ model is declared within a namespace, then use **SET_MODEL_NAMESPACE(model_namespace)** macro to specify the namespace. See \ModelBuilder\include\ModelBuilder.h in SystemVue installation directory for macro definition.
- The C++ code generator relies on the **names** specified through the **DFInterface** to use model's member variables in the generated code. Therefore, model member variables for inputs, outputs, parameters, and array parameter sizes must be in public scope, and the names of the member variables must be specified exactly the same as declared in the model class. The macros, e.g., **ADD_MODEL_INPUT(user_variable)**, **ADD_MODEL_OUTPUT(user_variable)**, **ADD_MODEL_PARAM(user_param_variable)**, **ADD_MODEL_ENUM_PARAM(user_param_variable, enum_type_name)**, **ADD_MODEL_ARRAY_PARAM(user_param_variable, user_array_size_variable)**, help users to add inputs, outputs, and parameters while preserving naming consistency. For advanced users, see *pcCodeGenName*, *pcSizeName*, and *pcEnumType* in \ModelBuilder\include\DFInterface.h and see \ModelBuilder\include\ModelBuilder.h in SystemVue installation directory.
- For enum parameters, the enum types must be declared in public scope. The names of enum types must be specified exactly the same as declaration and must include class scope if the enum types are declared within classes. See **ADD_MODEL_ENUM_PARAM(user_param_variable, enum_type_name)** macro in \ModelBuilder\include\ModelBuilder.h in SystemVue installation directory.

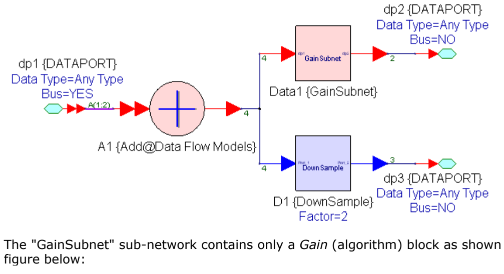
  > Model's inputs, outputs, parameters, array parameter sizes, and enum types must be declared in public scope, and the names and enum types must be specified properly.

- After code generation, the Visual Studio solution and projects (see Generating Code and Supported Targets) that are automatically created by SystemVue will have the proper include and library directories for the built-in SystemVue models. Regarding to custom (user-defined) C++ models, users have to **manually** include them in the Visual Studio projects. The following steps provide a general guideline to build the custom C++ models along with the generated code.
  1. Copy the custom .h and .cpp files to the generated Visual Studio project directory.
  2. In Visual Studio Solution Explorer, right click the project, use Add > Existing Files to add the custom .h and .cpp files to the project Header and Source Files.
  3. In the project property page (right click the project in Solution Explorer, then choose Properties), set the include directories (Configuration Properties > C/C++ > General > Additional Include Directories), library directories (Configuration Properties > Linker > General > Additional Library Directories), and .lib files (Configuration Properties > Linker > Input > Additional Dependencies) that are necessary to build the custom C++ models. See *Using Third Party Library in C++ Models* (users) for information about how to setup Visual Studio project for using third party libraries in C++ models.
  4. If the custom C++ models depend on dynamic link libraries, remember to set windows **PATH** environment variable to include the directory where the .dll files are located.

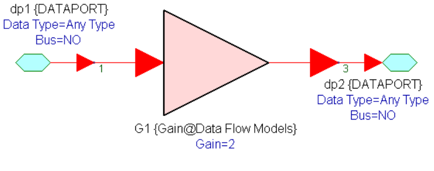
Understanding Generated C++ Code

Example

The following figure shows a sub-network example for C++ code generation. The sub-network contains an *Add* (algorithm) block *A1*, a "GainSubnet" sub-network model *Data1*, and a custom "DownSample" C++ model *D1*.



The "GainSubnet" sub-network contains only a *Gain* (algorithm) block as shown in the figure below:



The blocks *Add* (algorithm) and *Gain* (algorithm) use *circular buffers* (users) as inputs and outputs. The header files can be found in \ModelBuilder\include under SystemVue installation directory.

The custom *C++ model* (users) "DownSample" implements a simple down sampler. The implementation is shown in the following "DownSample.h" and "DownSample.cpp" for the purpose of illustrating scalar port (*double Out*) and array port (*double *In*).

```
// DownSample.h
```

```
#pragma once
#include "ModelBuilder.h"
class DownSample : public AgilentEEsof::DFModel
{
public:
 DECLARE_MODEL_INTERFACE( DownSample )
 virtual bool Run(); // down sampling
 virtual bool Setup(); // Setup rate
 double *In;  // array input
 double Out;  // scalar output
 int Factor;  // down sample factor
 unsigned Rate; // input rate = down sample factor
};


// DownSample.cpp
#include "stdafx.h"
#include "DownSample.h"
DEFINE_MODEL_INTERFACE( DownSample )
{
 AgilentEEsof::DFParam cFactor = ADD_MODEL_PARAM( Factor );
 cFactor.SetDefaultValue( "2" );
 AgilentEEsof::DFPort cIn = ADD_MODEL_INPUT( In );
 cIn.AddRateVariable( Rate );
 ADD_MODEL_OUTPUT( Out );
 return true;
}
bool DownSample::Setup()
{
 if ( Factor > 0 )
 Rate = Factor;
 else
 POST_ERROR("Factor should be > 0");
 return true;
}
bool DownSample::Run()
{
 Out = In[0];
 return true;
}
```

Generated Header and C++ Files

The following "MyModel.h" shows the generated C++ model header file for the above code generation sub-network.

- The top of the header file documents the file name and copyright notice.
- It includes the header files that declare the models inside the code generation sub-network.
- The class name of the generated C++ model is specified by the **Generated Class Name** field in C++ Code Generation Options dialogue box.
- For each sub-network interface port or bus-port, e.g., *dp1*, *dp2*, and *dp3* in Fig: GainSubnet, there is a corresponding *circular buffer* (users) port or *circular buffer bus* (users) port declared in the generated C++ model for data input and output.
- The hierarchical sub-networks are preserved in the generated model in a way that the models are declared in nested classes that imitate the hierarchical structures. For example, block *G1* in sub-network *Data1* is invoked in the generated code as *Data1.G1*. For example, block *A1* in the top-level code generation network is invoked simply as *A1*.
- If a model has any scalar port or array port, a *circular buffer* (users) will be declared with the model for accessing data in a circular buffer fashion. For example, circular buffers *D1_In* and *D1_Out* are declared for "DownSample" *D1.In* and *D1.Out*.
- For each connection in the code generation network, there is a corresponding buffer memory declared to store data for the connection. For example, double* m_pBuffer_Data1_G1_output_To_dp2 for connection from *Data1.G1.output* to *dp2*.

```
/*
* MyModel.h
* Created by SystemVue C++ Code Generator
* Copyright &copy; Agilent Technologies, Inc. 2000-2010
*/
#pragma once
#include "ModelBuilder.h"
#include "DownSample.h"
#include "SystemVue/Models/Gain.h"
#include "SystemVue/Models/Add.h"
#include "SystemVue/Models/Fork.h"
class MyModel : public AgilentEEsof::DFModel
{
public:
 DECLARE_MODEL_INTERFACE(MyModel)
 MyModel();
 ~MyModel();
 bool Setup();
 bool Initialize();
 bool Run();
 bool Finalize();
 // input, size=2, rate=2 2
 AgilentEEsof::CircularBufferBusT<AgilentEEsof::CircularBuffer<double > > dp1;
 // output, rate=1
 AgilentEEsof::CircularBuffer<double > dp3;
 // output, rate=2
 AgilentEEsof::CircularBuffer<double > dp2;
private:
 // subnetwork Data1
 class Subnetwork_Data1
 {
 public:
 // AgilentEEsof::Gain< double > double Gain
 AgilentEEsof::Gain< double > G1;
 } Data1;
 // delete buffer memory
 void DeleteBuffers();
 // DownSample
 DownSample D1;
 // circular buffer for D1.In
 AgilentEEsof::CircularBuffer<double > D1_In;
 // circular buffer for D1.Out
 AgilentEEsof::CircularBuffer<double > D1_Out;
 // AgilentEEsof::Add< double > double Add
 AgilentEEsof::Add< double > A1;
 // AgilentEEsof::Fork< AgilentEEsof::CircularBuffer< double > > double Fork
 AgilentEEsof::Fork< AgilentEEsof::CircularBuffer< double > > A1_output;
 // buffer from dp1[0] to A1.input[0]
 double* m_pBuffer_dp1_0__To_A1_input_0_;
 // circular buffer for dp1[0]
 AgilentEEsof::CircularBuffer<double > dp1_0__CirBuf;
 // buffer from dp1[1] to A1.input[1]
 double* m_pBuffer_dp1_1__To_A1_input_1_;
 // circular buffer for dp1[1]
 AgilentEEsof::CircularBuffer<double > dp1_1__CirBuf;
 // buffer from D1_Out to dp3
 double* m_pBuffer_D1_Out_To_dp3;
 // circular buffer for dp3
 AgilentEEsof::CircularBuffer<double > dp3_CirBuf;
 // buffer from Data1.G1.output to dp2
 double* m_pBuffer_Data1_G1_output_To_dp2;
 // circular buffer for dp2
 AgilentEEsof::CircularBuffer<double > dp2_CirBuf;
 // buffer from A1.output to A1_output.input
 double* m_pBuffer_A1_output_To_A1_output_input;
 // buffer from A1_output.output[0] to D1_In
 double* m_pBuffer_A1_output_output_0__To_D1_In;
 // buffer from A1_output.output[1] to Data1.G1.input
 double* m_pBuffer_A1_output_output_1__To_Data1_G1_input;
};
```

The following "MyModel.cpp" shows the generated C++ model cpp file for the above code generation sub-network.

- Input and output circular buffers and circular buffer buses are added automatically in *DEFINE_MODEL_INTERFACE* (users) such that it can be easily brought back to SystemVue. The *DEFINE_MODEL_INTERFACE* (users) is surrounded by **SV_CODE_GEN** such that it can be easily compiled out for standalone usage.
- Constructor, destructor, and *DeleteBuffers()* methods take care of initialization and

de-allocation of buffer memories.

- *Setup()* method is overridden to set model's parameters (if any), initialize model's bus-port width (if any), declare contiguous memory for model's array port (if any), set optional connectivity for model's circular buffer port (if any), and call each model's Setup() methods. It also initialize the interface circular buffer bus width and set the input and output data flow rates of the generated model.
- *Initialize()* method is overridden to allocate buffer memories based on the computed schedule and set circular buffers for both ends of the connections. It also calls each model's Initialize() methods.
- *Run()* method is overridden to read data from input circular buffer (bus) ports, execute the pre-computed schedule for a complete data flow iteration, and write data to output circular buffer (bus) ports. Before and after each model's Run() method, data access and circular buffer adjustment are taken care properly.
- *Finalize()* method is overridden to call each model's Finalize() method and to de-allocate buffer memories.

```
/*
* MyModel.cpp
* Created by SystemVue C++ Code Generator
* Copyright &copy; Agilent Technologies, Inc. 2000-2010
*/
#include "MyModel.h"
#ifndef SV_CODE_GEN
DEFINE_MODEL_INTERFACE(MyModel)
{
 ADD_MODEL_INPUT( dp1 );
 ADD_MODEL_OUTPUT( dp3 );
 ADD_MODEL_OUTPUT( dp2 );
 return true;
}
#endif
MyModel::MyModel()
{
 m_pBuffer_dp1_0__To_A1_input_0_ = NULL;
 m_pBuffer_dp1_1__To_A1_input_1_ = NULL;
 m_pBuffer_D1_Out_To_dp3 = NULL;
 m_pBuffer_Data1_G1_output_To_dp2 = NULL;
 m_pBuffer_A1_output_To_A1_output_input = NULL;
 m_pBuffer_A1_output_output_0__To_D1_In = NULL;
 m_pBuffer_A1_output_output_1__To_Data1_G1_input = NULL;
}
MyModel::~MyModel()
{
 DeleteBuffers();
}
void MyModel::DeleteBuffers()
{
 delete[] m_pBuffer_dp1_0__To_A1_input_0_;
 m_pBuffer_dp1_0__To_A1_input_0_ = NULL;
 delete[] m_pBuffer_dp1_1__To_A1_input_1_;
 m_pBuffer_dp1_1__To_A1_input_1_ = NULL;
 delete[] m_pBuffer_D1_Out_To_dp3;
 m_pBuffer_D1_Out_To_dp3 = NULL;
 delete[] m_pBuffer_Data1_G1_output_To_dp2;
 m_pBuffer_Data1_G1_output_To_dp2 = NULL;
 delete[] m_pBuffer_A1_output_To_A1_output_input;
 m_pBuffer_A1_output_To_A1_output_input = NULL;
 delete[] m_pBuffer_A1_output_output_0__To_D1_In;
 m_pBuffer_A1_output_output_0__To_D1_In = NULL;
 delete[] m_pBuffer_A1_output_output_1__To_Data1_G1_input;
 m_pBuffer_A1_output_output_1__To_Data1_G1_input = NULL;
}
bool MyModel::Setup()
{
 bool bStatus = true;
 //setup models
 //DownSample D1
 D1.Factor = 2;
 D1.Phase = 0;
 D1_In.SetContiguousProperty();
 bStatus &= D1.Setup();
 //AgilentEEsof::Gain< double > Data1.G1
 Data1.G1.m_Gain = 2;
 bStatus &= Data1.G1.Setup();
 //AgilentEEsof::Add< double > A1
 A1.input.Initialize(2);
 bStatus &= A1.Setup();
 //AgilentEEsof::Fork< AgilentEEsof::CircularBuffer< double > > A1_output
 A1_output.output.Initialize(2);
 bStatus &= A1_output.Setup();
 //setup circular buffer buses
 dp1.Initialize(2);
 //setup input and output dataflow rates
 dp1[0].SetRate(2);
 dp1[1].SetRate(2);
 dp3.SetRate(1);
 dp2.SetRate(2);
 return bStatus;
}
bool MyModel::Initialize()
{
 bool bStatus = true;
 DeleteBuffers();
 //allocate buffer from dp1[0] to A1.input[0]
 m_pBuffer_dp1_0__To_A1_input_0_ = new double[2];
 dp1_0__CirBuf.SetBuffer(m_pBuffer_dp1_0__To_A1_input_0_, 2, 2, 0);
 A1.input[0].SetBuffer(m_pBuffer_dp1_0__To_A1_input_0_, 2, 1, 0);
 //allocate buffer from dp1[1] to A1.input[1]
 m_pBuffer_dp1_1__To_A1_input_1_ = new double[2];
 dp1_1__CirBuf.SetBuffer(m_pBuffer_dp1_1__To_A1_input_1_, 2, 2, 0);
 A1.input[1].SetBuffer(m_pBuffer_dp1_1__To_A1_input_1_, 2, 1, 0);
 //allocate buffer from D1_Out to dp3
 m_pBuffer_D1_Out_To_dp3 = new double[1];
 dp3_CirBuf.SetBuffer(m_pBuffer_D1_Out_To_dp3, 1, 1, 0);
 D1_Out.SetBuffer(m_pBuffer_D1_Out_To_dp3, 1, 1, 0);
 //allocate buffer from Data1.G1.output to dp2
 m_pBuffer_Data1_G1_output_To_dp2 = new double[2];
 dp2_CirBuf.SetBuffer(m_pBuffer_Data1_G1_output_To_dp2, 2, 2, 0);
 Data1.G1.output.SetBuffer(m_pBuffer_Data1_G1_output_To_dp2, 2, 1, 0);
 //allocate buffer from A1.output to A1_output.input
 m_pBuffer_A1_output_To_A1_output_input = new double[1];
 A1.output.SetBuffer(m_pBuffer_A1_output_To_A1_output_input, 1, 1, 0);
 A1_output.input.SetBuffer(m_pBuffer_A1_output_To_A1_output_input, 1, 1, 0);
 //allocate buffer from A1_output.output[0] to D1_In
 m_pBuffer_A1_output_output_0__To_D1_In = new double[2];
 A1_output.output[0].SetBuffer(m_pBuffer_A1_output_output_0__To_D1_In, 2, 1, 0);
 D1_In.SetBuffer(m_pBuffer_A1_output_output_0__To_D1_In, 2, 2, 0);
 //allocate buffer from A1_output.output[1] to Data1.G1.input
 m_pBuffer_A1_output_output_1__To_Data1_G1_input = new double[1];
 A1_output.output[1].SetBuffer(m_pBuffer_A1_output_output_1__To_Data1_G1_input, 1, 1, 0);
 Data1.G1.input.SetBuffer(m_pBuffer_A1_output_output_1__To_Data1_G1_input, 1, 1, 0);
 //initialize models
 bStatus &= D1.Initialize();
 bStatus &= Data1.G1.Initialize();
 bStatus &= A1.Initialize();
 bStatus &= A1_output.Initialize();
 return bStatus;
}
bool MyModel::Run()
{
 bool bStatus = true;
 //copy samples from inputs
 dp1[0].Copy(0, &dp1_0__CirBuf, 0, 2);
 dp1[1].Copy(0, &dp1_1__CirBuf, 0, 2);
 //loop indices
 int index1;
 //execute schedule
 for (index1=0; index1<2; index1++ ) {
 //AgilentEEsof::Add< double > A1
 bStatus &= A1.Run();
 A1.input[0].Advance();
 A1.input[1].Advance();
 //AgilentEEsof::Fork< AgilentEEsof::CircularBuffer< double > > A1_output
 bStatus &= A1_output.Run();
 A1_output.output[0].Advance();
 //AgilentEEsof::Gain< double > Data1.G1
 bStatus &= Data1.G1.Run();
 Data1.G1.output.Advance();
 }
```

```
 //DownSample D1
 D1.In = (double*)D1_In.GetReadPtr();
 bStatus &= D1.Run();
 D1_Out[0] = D1.Out;
 //copy samples to outputs
 dp3_CirBuf.Copy(0, &dp3, 0, 1);
 dp2_CirBuf.Copy(0, &dp2, 0, 2);
 return bStatus;
}
bool MyModel::Finalize()
{
 bool bStatus = true;
 //finalize models
 bStatus &= D1.Finalize();
 bStatus &= Data1.G1.Finalize();
 bStatus &= A1.Finalize();
 bStatus &= A1_output.Finalize();
 DeleteBuffers();
 return bStatus;
}
```

Generated Code and SystemVue Sub-network Differences

In most cases the generated code will behave exactly the same as the SystemVue sub-network it was generated from. This section lists some exceptions:

1. All anytype models (models with red ports) are replaced (in the generated code) by specific type models. In the example described in this section, the anytype gain and add models are being replaced by gain and add models that operate on double numbers, since double was the resolved type for these models. If the resolved type for these models were complex, then they would be replaced by gain and add models that operate on complex numbers. The generated code can only operate on specific data types and once generated the data type cannot be changed when the generated code is being used (run). Of course, the data type can be changed if the code is generated again with a different set of input signals or parameters, which result in a different resolved type for the anytype models.
2. For improved performance, certain models are being replaced by simpler more efficient versions and therefore the generated code does not have the full functionality of the SystemVue sub-network it was generated from. For example, the Math model is replaced by a model that performs only the specific function selected during code generation, e.g. Sqrt. Therefore, if the FunctionType parameter of the Math model was controlled by a parameter of the top level sub-network, the model that replaces the Math model in the generated code would not respond to changes of this top level sub-network parameter. The following table lists all the models for which the generated code will not have the full functionality of the original model.

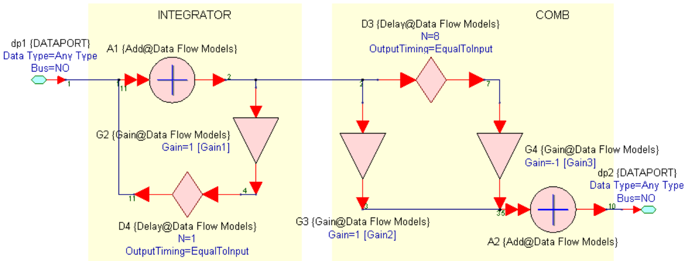
Original Model	**Generated Model**
Math (algorithm)	performs only the function selected during code generation (FunctionType parameter is removed)
MathCx (algorithm)	performs only the function selected during code generation (FunctionType parameter is removed)
Trig (algorithm)	performs only the function selected during code generation (FunctionType parameter is removed)
TrigCx (algorithm)	performs only the function selected during code generation (FunctionType parameter is removed)
Logic (algorithm)	performs only the function selected during code generation (Logic parameter is removed)
RandomBits (algorithm)	does not have bust capability if BurstMode was set to OFF (all Burst related parameters are removed)
PRBS (algorithm)	does not have bust capability if BurstMode was set to OFF (all Burst related parameters are removed)
DataPattern (algorithm)	does not have bust capability if BurstMode was set to OFF (all Burst related parameters are removed)
WaveForm (algorithm)	does not have bust capability if BurstMode was set to OFF (all Burst related parameters are removed)

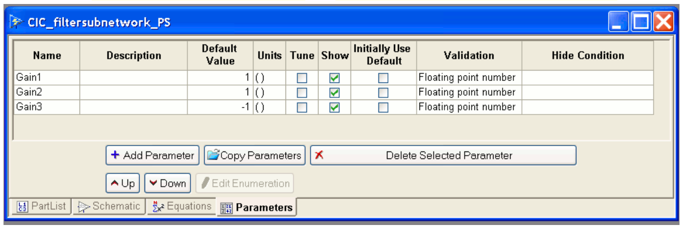
3. See next section on Parameter Support for other cases where the generated code may not behave the same as the SystemVue sub-network it was generated from.

Parameter Support

When a sub-network is selected for code generation and the sub-network has parameters, the C++ code generator will create corresponding public members in the generated class, which can be used to parametrize and control the model. To enable parametrization in the generated code, at least one of the parts inside the sub-network must make use of the sub-network parameters to set its own parameters.

The following figure shows a CIC filter sub-network, where the *Gain* parameters of the *Gain* (algorithm) parts are set by the sub-network parameters *Gain1*, *Gain2*, and *Gain3*.





The generated model, *MyCICPS*, for the above CIC filter sub-network is partially shown in the following code. In the class declaration three parameters *Gain1*, *Gain2*, and *Gain3* are declared as double (this depends on the **Validation** flag in the sub-network Parameters tab) variables. In DEFINE_MODEL_INTERFACE, *Gain1*, *Gain2*, and *Gain3* are added as parameters of the generated model. In Setup, the m_Gain members of the AgilentEEsof::Gain< double > models *G2*, *G3*, and *G4* are set using *Gain1*, *Gain2*, and *Gain3*.

```
/*MyCICPS.h*/
class MyCICPS : public AgilentEEsof::DFModel
{
public:
 //sub-network parameters
 double Gain1;
 double Gain2;
 double Gain3;
 //...
private:
 // AgilentEEsof::Gain< double > double Gain
 AgilentEEsof::Gain< double > G2;
```

```
 // AgilentEEsof::Gain< double > double Gain
 AgilentEEsof::Gain< double > G3;
 // AgilentEEsof::Gain< double > double Gain
 AgilentEEsof::Gain< double > G4;
 //...
};
/*MyCICPS.cpp*/
#ifndef SV_CODE_GEN
DEFINE_MODEL_INTERFACE(MyCICPS)
{
 ADD_MODEL_PARAM(Gain1);
 ADD_MODEL_PARAM(Gain2);
 ADD_MODEL_PARAM(Gain3);
 //...
}
#endif
bool MyCICPS::Setup()
{
 //...
 //AgilentEEsof::Gain< double > G2
 G2.m_Gain = Gain1;
 bStatus &= G2.Setup();
 //AgilentEEsof::Gain< double > G3
 G3.m_Gain = Gain2;
 bStatus &= G3.Setup();
 //AgilentEEsof::Gain< double > G4
 G4.m_Gain = Gain3;
 bStatus &= G4.Setup();
 //...
}
```

The mapping between the **Validation** flag of a sub-network parameter and the type of the C++ variable created is shown in the table below:

Validation Flag	**C++ variable type**
Boolean	bool
Integer	int
Positive Integer	int
Floating point number	double
Warn if negative	double
Warn if non-positive	double
Error if negative	double
Error if non-positive	double
Complex number	std::complex<double>
Integer array	AgilentEEsof::Matrix<int>
Floating point array	AgilentEEsof::Matrix<double>
Complex array	AgilentEEsof::Matrix< std::complex<double> >
Enumeration	int
Text	char*
Filename	char*
Warning	NOT SUPPORTED
Error	NOT SUPPORTED
<None>	NOT SUPPORTED

> ⚠ For code generation purposes, users MUST properly set the **Validation** flag for each sub-network parameter in the sub-network Parameter tab.

> ⚠ In the SystemVue 2010.07 release the parameter support is limited to direct assignments (e.g. Gain=Gain1) of the sub-network parameters to the parameters of its parts (see CIC filter examples described earlier in this section). If a part is using a sub-network model then again the parts inside that sub-network can only use the sub-network's parameters in direct assignments to set their parameters. There is no limit to the number of hierarchy levels supported. For example, let *A* be the top level sub-network that is selected for code generation and *a* be a parameter of *A*. Let *B* be a sub-network inside *A* and *b* be a parameter of *B* set to *a*. Let *C* be a part (not using a sub-network model) inside *B* and *c* be a parameter of *C* set to *b*. Then in the generated code, *B.C.c* is set to *a* and so changes to the top level sub-network parameter *a* are properly propagated to the lower hierarchy levels.

> ⚠ If a part's parameter is set using an expression or equation, then in the generated code the parameter will be set to the expression's resolved value and changing the values of the top level sub-network's parameters will have no effect on the behavior of the part. For example, in the CIC filter example shown earlier in this section, if the Gain parameter of part G2 is set to 2*Gain1-0.3, the generated code will set G2.m_Gain to 2*1-0.3=1.7 and changing Gain1 will not affect the Gain of part G2.

> ⚠ When a sub-network parameter is used in a direct assignment to set the parameter of one of its parts and it is also used in an expression to set the parameter of another one of its parts, users must be aware of the inconsistency in the behavior of the generated model. For example, let *p* be a top level sub-network parameter. Let *X* be a part inside the top level sub-network whose parameter *x* is set to *p*. Let *Y* be a part inside the top level sub-network whose parameter *y* is set to 1_*p*. Then in the generated code, *X.x* is set to *p* and thus controlled by it, whereas *Y.y* is set to the resolved value of the 1_*p* and cannot be controlled by *p*. In this case, the generated model will not behave the same as the original sub-network model when the value of the parameter *p* is changed.

> ⚠ If a part's parameter can change the data flow rate or buffer size or fixed point parameters of the part's input/output, setting the part's parameter by the sub-network parameter may introduce incorrect behavior in the generated model. This is because the schedule, the buffer size, and the fixed point parameters in the generated model are pre-computed and hard-coded based on the parameter values during code generation. In this case, when such parameter is changed from its default value, incorrect behavior may occur in the generated model.

> ⚠ In certain cases, model parameters are removed from the generated code (see section Generated Code and SystemVue Sub-network Differences). Trying to control these parameters with a sub-network parameter is going to result in inconsistent behavior between the original SystemVue sub-network and the generated code.

Limitations

- The data flow graph inside the code generation network (sub-network) must be connected. The C++ code generator does not support multiple isolated graphs because the relative execution rates depend on outside systems.
- The C++ code generator currently does not support *timed* (sim) blocks, *envelope* (sim) blocks, nor *dynamic* (sim) blocks.

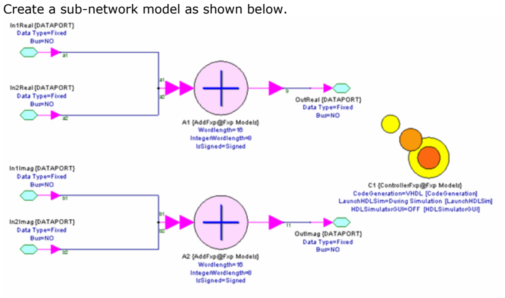
HDL Code Generation

SystemVue provides its users with an easy path from schematic design to the hardware. This could be done by using HDL Code Generation capability of SystemVue. A user created SystemVue *sub-network model* (users), using only synthesizeable Fixed Point parts from *Hardware Design Library* (hardware) can be used to generate VHDL/Verilog for the sub-network.

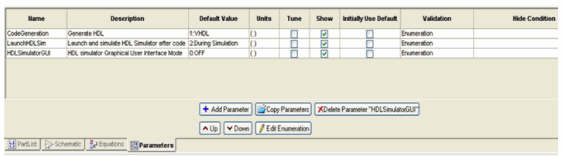
In this tutorial we will go through a simple example to understand the design flow to generate HDL Code. The same design flow can be used for more complex designs. We will create a Fixed point design for a Complex Adder, generate VHDL for the Complex Adder and performs functional verification of the generated VHDL.

Generating Fixed Point Sub-Network Model

- If you have note done so then read and understand *Sub-Network Models* (users) documentation.
- Create a sub-network model as shown below.



- Note that ControllerFxp component is included in the design. Also note that the values of parameters CodeGeneration, LaunchHDLSim and HDLSimulatorGUI are assigned the model parameter names.
- Set the Sub-Network parameters as follows

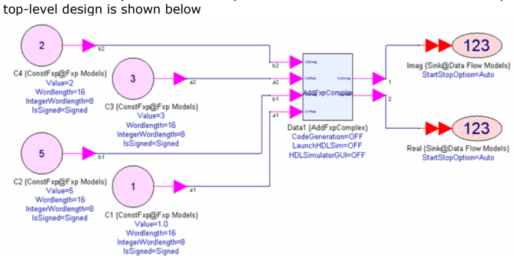


Name	Description	Default Value	Units	Tune	Show	Initially Use Default	Validation	Hide Condition
CodeGeneration	Generate HDL	1:VHDL	()	☐	☑	☐	Enumeration	
LaunchHDLSim	Launch and simulate HDL Simulator after code	2:During Simulation	()	☐	☑	☐	Enumeration	
HDLSimulatorGUI	HDL simulator Graphical User Interface Mode	0:OFF	()	☐	☑	☐	Enumeration	

- For the *sub-network model* (users) parameters, edit the enumeration type for *CodeGeneration* to be *ControllerFxp_CodeGeneration*, for *LaunchHDLSim* to be *ControllerFxp_LaunchHDLSim* and for *HDLSimulatorGUI* to be *ConstrollerFxp_HDLSimulatorGUI* from the library *Fxp Enums*.

> Parameterizing the sub-network model to control ControllerFxp is not required but helps if you would like to use the same sub-network model for Fixed point simulation and automatic HDL Cosimulation.

- Create a top level design and simulate the sub-network model to make sure its correct functionality. Make sure that parameter *CodeGeneration=OFF*. An example top-level design is shown below



> **Warning**
> The fixed point sub-network model used to generate HDL must contains only synthesizeable Fixed Point parts from *Hardware Design Library* (hardware). For example *FloatToFxp* (hardware) part is not synthesizeable. To ensure that a Fixed Point part is synthesizeable, consult its documentation.

Generating the HDL and HDL Simulation

- Change the value of model parameter *CodeGeneration* to *VHDL*. If you have not parameterized your sub-network model then select *CodeGeneration=VHDL* for ControllerFxp component inside the sub-network.
- Simulate the top level design, it will perform the fixed point simulation and generates the corresponding *VHDL* for the sub-network model instance in the sub-directory ***<schematic design name>_<sub-network model instance name>_HDL\hdl*** under the same directory where the workspace containing the design is located. The name of the file containing the top level VHDL will be ***<sub-network model name>.vhd***.

If you have installed ModelSim SE and it is configured to run from command line i.e. your operating system's PATH variable points to ModelSim SE, then you can also invoke HDL simulation **After Simulation** using the test vectors generated by SystemVue simulation. Alternatively you can use automatic *HDL Cosimulation* (sim) **During Simulation** to make sure that generated VHDL is functionally correct. The automatic *HDL Cosimulation* (sim), performs inter process communication between SystemVue and ModelSim to process data by HDL simulator in real time while SystemVue simulation is running. The Hdl simulation **During Simulation** first generate the HDL for the sub-network and then runs the HDL portion of the design in ModelSim and rest in SystemVue using inter process communication.

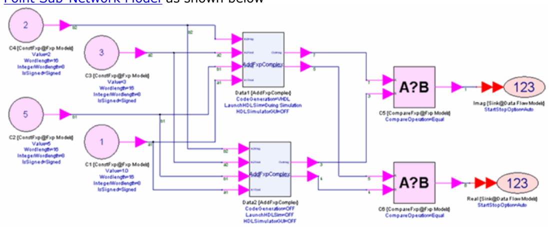
- Change the value of model parameter *LaunchHDLSim=After Simulation* and run the simulation. This will simulate the design in SystemVue and at the end of simulation starts ModelSim and run ModelSim simulation using the test vectors generated by the SystemVue simulation.
- To perform HDL simulation **During Simulation** change the value of model parameter *LaunchHDLSim=During Simulation* and run the simulation. This will perform the HDL Cosimulation in the back ground if *HDLSimulatorGUI=OFF*.
- Now change the value of model parameter *HDLSimulatorGUI=ON* and run the simulation. This will bring up ModelSim GUI and halts the SystemVue simulation which is waiting for data from ModelSim. This interactive mode can be used for debugging HDL code. To resume the simulation, either issue the command **run** inside ModelSim to run for a single step or **run -all** to run the complete simulation.

> - Currently only VHDL is supported with *LaunchHDLSim=During Simulation*.
> - If *HDLSimulatorGUI=ON* then close the ModelSim before starting simulation again. Otherwise you will require additional license for new instance of ModelSim.

Testing for Functional Equivalency

This section covers that how to perform functional equivalency test to prove that SystemVue generated HDL is functionally equivalent to the fixed point model for which HDL was generated. You must have ModelSim SE installed properly, see section Generating the HDL and HDL Simulation for more details.

- Create a design using the sub-network model created in section Generating Fixed Point Sub-Network Model as shown below



- This design includes two instances of the sub-network model. One instance is configured to generate VHDL and LaunchHDLSimulation=During Simulation, the other instance is configured not to generate HDL and perform only fixed point simulation.
- Both instances are fed with the same inputs.
- The output of both instances are compared using ***CompareFxp* (hardware)** part with CompareOperation=Equal. The output of ***CompareFxp* (hardware)** is '1' to indicate true and is '0' to indicate false.
- Run the simulation, and observe the results. If the output of all *CompareFxp* (hardware) instances is always '1' that means the generated HDL is functionally equivalent to original fixed point model.

Warning
- Currently only VHDL can be used with *LaunchHDLSimulation=During Simulation* therefore it is not possible to perform functional equivalency test for SystemVue generated Verilog.
- If you are using any delay or changing the sample rate in the sub-network model then you cannot use any downstream component that is backward reachable simultaneous to a sub-network model instances using *LaunchHDLSimulation=During Simulation* and sub-network model instances **not** using *LaunchHDLSimulation=During Simulation*. The functional verification can still be performed by connecting the output of sub-network model instances to two different sinks and comparing the data in the dataset. In the above example it means not using the *CompareFxp* (hardware).

The reason that outputs of sub-network model instances (one with HDL Cosimulation and the other with fixed-point simulation) cannot be combined at the input of a downstream component is because sub-network model with LaunchHDLSimulation=During Simulation is replaced by a single *HdlCosim* (sim) model which is a uni-rate model, where as the other instance uses the fixed-point models in the simulation.

Understanding the Generated HDL

The generated HDL will be located inside the sub-directory ***<schematic design name>_<sub-network model instance name>_HDL\hdl*** under the same directory where the workspace containing the design is located. The name of the file containing the top level HDL will be ***<sub-network model name>.vhd*** for VHDL and ***<sub-network model name>.v*** for verilog.
The number of input/output ports will be same if there is no sequential component is used, for example the Complex Adder design above. However, in case of using sequential components such as *DelayFxp* (hardware), *RegisterFxp* (hardware) etc, the resulting HDL will have extra **CLK** and **RESET** input ports, it may also have a **DataInEnable** input port as well, depending upon the design. This **DataInEnable** control input port must be used to indicate when the input data is valid ('1') and when it is not ('0'). There may be a **DataOutEnable** output port which may be used to detect when the output of the HDL is valid ('1') and when it is not valid ('1').
Other than the top level HDL file other HDL files are also included. Most of these HDL files contains HDL for sub-components used to create the top level HDL and must be included in any synthesis/simulation tool along with the top-level HDL.

SystemVue Examples

SystemVue ships with hardware design examples which are configured or can be configured to generate HDL. These examples are installed under **<SystemVue install directory>\Examples\Hardware Design**.

IBIS-AMI Model Generation

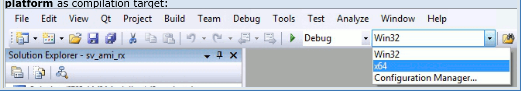
SystemVue IBIS Algorithmic Model Interface (IBIS-AMI) Generation allows users to generate IBIS-AMI models for a sub-network of a system. The generated code conforms to IBIS Version 5.0 and can be used with EDA Platforms (channel simulators) which supports IBIS-AMI models simulation. A Visual Studio project is automatically created for generating AMI models dll on Microsoft Windows platform. The IBIS-AMI Model is created as a target configuration in *C++ Code Generation* (users).

> The automatically generated Visual Studio project for Microsoft Windows supports building AMI model DLL's for **both 32 bit and 64 bit** platforms.

Requirements

- SystemVue must be installed on the machine where you will be building the IBIS-AMI Model on Windows platform.
- Building IBIS-AMI Models on Windows platform also requires either:
  - **Microsoft Visual C++ 2008 Express Edition** (under the **Visual Studio 2008 Express** tab)
  - **Microsoft Visual Studio 2008 with SP1** or **Microsoft Visual Studio C++ 2008 with SP1**

  > You **MUST** have a **FULL** installation of ***Microsoft Visual Studio 2008*** in order to compile for 64 bit target platform.

  > **The following screen capture of Microsoft Visual Studio shows how to choose 64 bit platform** as compilation target:
  >
  > 
  >

Licensing

- **If you have W1714F, the demo version of IBIS-AMI model generation:**
  - You will be able to simulate models in *IBIS-AMI Transceiver* (algorithm) category under "Algorithmic Design" library, and any other model for which you are licensed.
  - You will be able to generate "IBIS-AMI" models and use those in a EDA platform.
  - The generated "IBIS-AMI" models can only be used on a machine where SystemVue is installed and all licenses are available which are required to simulate the original sub-network in SystemVue.
- **If you have W1714, the full version of IBIS-AMI model generation:**
  - You will be able to simulate models in *IBIS-AMI Transceiver* (algorithm) category under "Algorithmic Design" library, and any other model for which you are licensed.
  - You will be able to generate "IBIS-AMI" models and use those in a EDA platform.
  - The generated "IBIS-AMI" models will not require a license if the original sub-network is designed only with models in **IBIS-AMI Transceiver** and **C++ Code Generation** categories under "Algorithmic Design" library and with the models available with W1718 license. These generated AMI model can be used anywhere in any EDA platform without needing a license during run time.

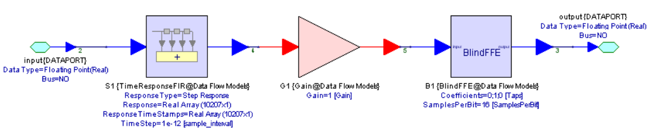
Prerequisite

Make sure that you have read and understood the documentation for *C++ Code Generation* (users), at least the *Quick Start* (users) section. You must be familiar with how to create a sub-network and to add that to *C++ Code Generator* (users) dialogue box. You must also know that how to select a **Target** in *C++ Code Generator* (users) as IBIS-Model generation uses an specific target **IBIS Algorithmic Modeling Interface**.

Also make sure that you have read and understood Section 6c and Section 10 of IBIS (I/O Buffer Information Specification) Version 5.0 explaining Algorithmic Model Interface specification.

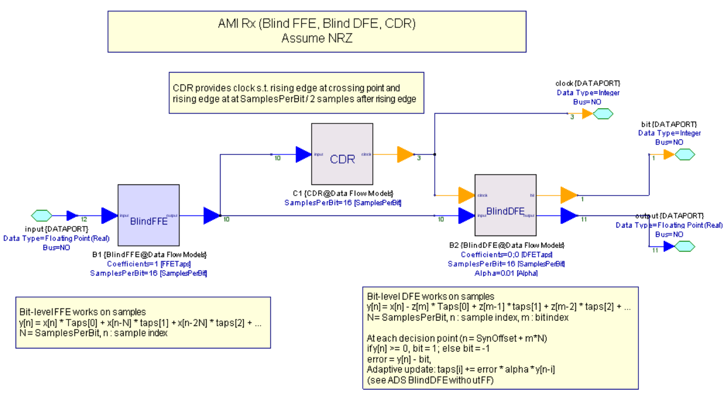
Creating AMI Sub-Network Models

The first step in generating IBIS-AMI model is to create a transmitter or receiver sub-network. An example transmitter sub-network is shown below



Please note that *TimeResponseFIR* (algorithm) filter model shown in the above sub-network is very useful in AMI modeling as it allows to use measured step or impulse response either using H-spice or a measurement instrument and re-characterize it to an specific TimeStep.

An example receiver sub-network is shown below



You can assign parameters to the sub-network and those will be available as either **Model_Specific** parameters or can be mapped to some of the input parameters of **AMI_Init** as defined in IBIS-AMI standard and can be accessed inside EDA platform.

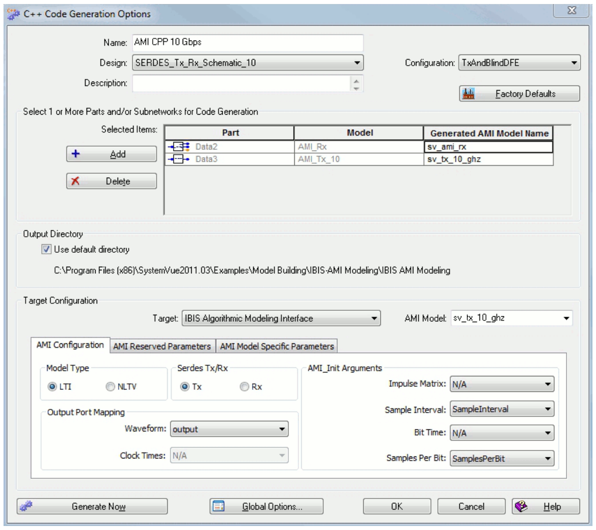
Each of the above sub-network will be exported as a separate AMI model to be used in an EDA platform.

Testing Sub-Network Models

Create a top level design to test your sub-network models. Simulate and make sure that you are satisfied with the functionality of the models.

Configuring Code Generator for AMI Models Generation

- Add a *C++ Code Generator* (users).
- Add the AMI sub-network models for code generation.
- Select Target in the C++ Code Generator to be **IBIS Algorithmic Modeling Interface**. After selecting the target the C++ Code Generations Options dialogue will be as shown below



Each field in this configuration windows is explained below

Generated AMI Model Name

This will be the model name of the generated AMI model for a particular sub-network.

AMI Model

This combo box contains all the Models which are added using Add button.You can select the AMI model whose export configuration needs to be specified. You can configure all models by selecting them one by one from this combo box.

AMI Configuration

In the AMI Configuration tab you can specify following options

Model Type

An exported model can be either Linear Time Invariant (LTI) or a Non-linear or Time Variant (NLTV) system. You can specify what kind of system your original sub-network implements. This is an important option because for an LTI model **AMI_Getwave** function is not implemented and the overall impulse response of the model is computed in the generated **AMI_Init** function. This means that for an AMI model, whose original sub-network model is an LTI system, the EDA platform will perform fast statistical computation by operating on the computed impulse response of the model. For an NLTV model, **AMI_Getwave** function is implemented and overall impulse response is not computed, instead sample by sample computation is performed by EDA platform by calling **AMI_Getwave** function.

Serdes Tx/Rx

Selects that the model is a transmitter (Tx) or receiver (Rx) model.

Output Port Mapping

You can have multiple output ports but you can select only two of these ports as output of your AMI model as explained below

- **Waveform:** Choose the output port which generates output wave samples of your AMI Model.
- **Clock Times:** Choose the output port which generates clock times. This mapping is available only if you are exporting an NLTV Rx model.

⚠ For AMI generation, the sub-network must have one and only one input port.

AMI_Init Arguments

As specified in Section 10 of IBIS Version 5.0 the **AMI_Init** function has the following function signature.

```
long AMI_Init (
 double *impulse_matrix,
 long row_size,
 long aggressors,
 double sample_interval,
 double bit_time,
 char *AMI_parameters_in,
 char **AMI_parameters_out,
 void **AMI_memory_handle, char **msg)
```

You can map selected arguments (impulse_matrix, sample_interval, bit_time) to your sub-network parameters. A EDA platform passes these arguments to AMI_Init, and if you map these to your corresponding sub-network parameters, the generated AMI_Init will map these arguments to the corresponding sub-network parameters. For convenience a short description of these arguments is given below

- **Impulse Matrix** (impulse_matrix argument) is the channel impulse response matrix, the impulse values are in volts and are uniformly spaced in time; the time spacing is given by **Sample Interval** explained next. The first column is the impulse response for the primary channel. The rest of the columns are the impulse responses from aggressor drivers to the victim receiver.

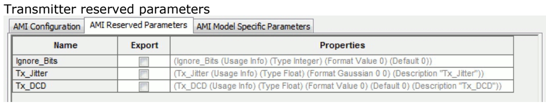
  ⚠ The generated AMI_Init will only pass the first column, i.e., the primary channel impulse response, to the corresponding sub-network parameter.
- **Sample Interval** (sample_interval argument) is the sampling interval of the channel impulse response.
- **Bit Time** (bit_time argument) is the unit interval (UI) of the current bit stream.
- **Samples Per Bit** is a parameter automatically calculated inside AMI_Init by

bit_time / sample_interval.

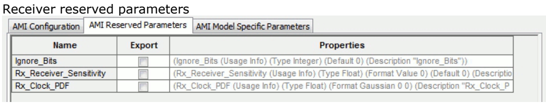
AMI Reserved Parameters

In the AMI Reserved Parameters as shown below you can specify which of the following AMI reserved parameters are exported as part of your AMI model (in the generated ami file).

Transmitter reserved parameters

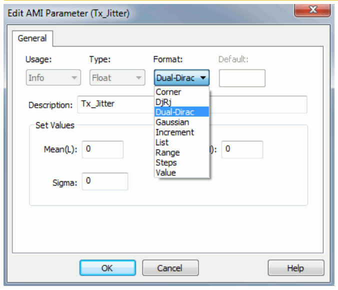


Receiver reserved parameters



To export a parameter select the **Export** check box and then click inside the **Properties** column for that parameter. This will open a dialogue box as shown below.

> ⚠ Make sure to select the **Format** that applies to the **Type** of the parameter as specified in IBIS specification.



Based on the **Format** type you can enter the values in this dialogue box and click **Ok**. This will generate an AMI formatted string and will be shown in **Properties** column for that parameter. This string will be included in the generated ami file.

Currently following selected reserved parameters are supported, and they are optional reserved parameters. The detailed description of these parameters is available in Section 6c of [IBIS Version 5.0](#) . A short description is given below for convenience.

Ignore Bits

Ignore Bits tell the EDA platform that how long the time variable (NLTV) model takes to complete initialization. This parameter should be exported only if you have selected [sv2011:Model Type](#) to be NLTV.

Tx_Jitter

Tells the EDA platform how much jitter exists at the input to the transmitter's analog output buffer.

TX_DCD

Tells the EDA platform the maximum percentage deviation of the duration of a transmitted pulse from the nominal pulse width.

RX_Clock_PDF

Tells the EDA platform the probability density function of the recovered clock.

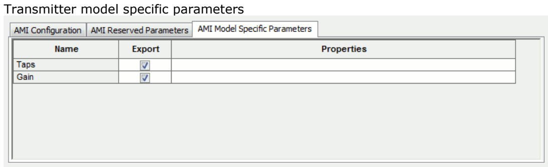
Rx_Receiver_Sensitivity

Tells the EDA platform the voltage needed at the receiver data decision point to ensure proper sampling of the equalized signal.

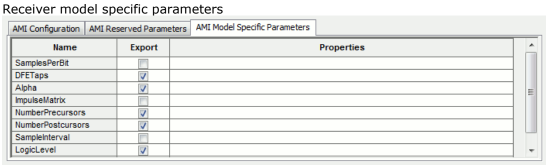
AMI Model Specific Parameters

Model specific parameters are extracted from the parameters in the underlying model. To export a parameter, similar to [**AMI Reserved Parameters**](#), select the **Export** check box and then click inside the **Properties** column for that parameter. This will open the parameter editor dialogue box.

Transmitter model specific parameters



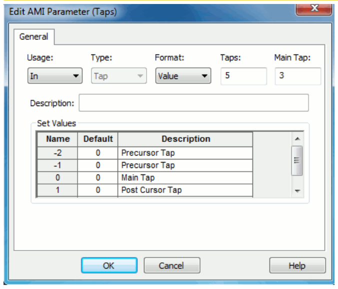
Receiver model specific parameters



Editing AMI Parameter of "Tap" Type

Most of the content in the AMI parameter editor dialog maps directly to the IBIS AMI specification. We will explain a few items related to Tap type here.
Taps specifies the number of taps required.
Main Tap specifies which tap (start from 1) is the main tap, such that any taps before it will be "Precursor" taps with negative indexing, while any taps following it will be "Post Cursor" taps with positive indexing as described in IBIS AMI specification.
In the example here, there are 5 taps total and the main tap is the 3rd tap, hence the indexing is: -2, -1, 0, 1 and 2.

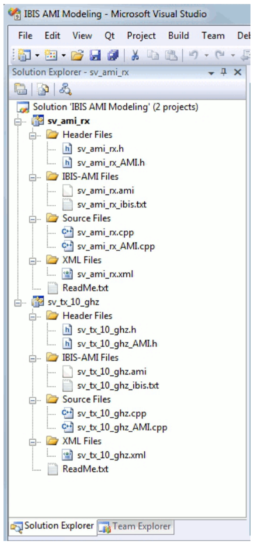
If you accidentally type in a"Main Tap:" value that is greater than the "Taps:" value, the "Main Tap:" value will be set to the same as "Tap:" value, which means the last tap will be the main tap, while all others are precursors.



Generating AMI Models

After sv2011:Configuring Code Generator for AMI Models Generation, click on **Generate Now** button to generate IBIS-AMI models. This will generate IBIS-AMI models in the **AMI** sub-directory of **Output Directory** or **Default Directory** as you have specified in Code Generator settings. A Visual Studio solution will be created with one project per AMI model.

The following picture shows the generated Visual Studio solution and projects for the "IBIS AMI Modeling" example workspace, using the "AMI CPP 10 Gbps" code generator in the workspace. This example can be found in \Examples\Model Building\IBIS-AMI Modeling under SystemVue installation directory.



As shown in the picture, for each AMI model, there is a corresponding Visual Studio project (with the same name as the AMI model) in the Visual Studio solution. Inside the project, "Header Files" and "Source Files" folders contain .h and .cpp files for the generated *C++ data flow sub-network model* (users) (e.g., sv_ami_rx.h and sv_ami_rx.cpp) and also contain the .h and .cpp files for the AMI interface functions, **AMI_Init**, **AMI_GetWave**, and **AMI_Close** (e.g., sv_ami_rx_AMI.h and sv_ami_rx_AMI.cpp). "IBIS-AMI Files" folder contains the generated AMI file (e.g., sv_ami_rx.ami) and a text file that contains the Algorithmic Model strings to be copied to the IBIS file (e.g., sv_ami_rx_ibis.txt).

You can build the whole solution either in Release or Debug mode and a <ami_model>.dll will be created per project. The generated <ami_model>.dll along with corresponding <ami_model>.ami file can be used inside EDA platform (channel simulator) to use this model, where <model_name> is the name of the AMI model specified in code generator dialogue box. Please note that <ami_model>.dll may or may not require corresponding SystemVue licensing when run inside the channel simulator based on what license you have when generating AMI model as explained in sv2011:Licensing section above.

Understanding AMI Model Generation

To understand the generated C++ data flow model for an AMI sub-network, please refer to *C++ Code Generation* (users).

The generated AMI (.ami) file conforms with Section 6.c in the IBIS Version 5.0 specification, and the generated AMI interface functions conform with Section 10 in the IBIS Version 5.0 specification.

For each generated AMI model (Visual Studio project), there is an IBIS-related text file (e.g., sv_ami_rx_ibis.txt) specifying compiler information, the name of the AMI model DLL (dynamic-link library), and the name of the AMI file. See "sv_ami_rx_ibis.txt" below as an example. The content should be copied to the **Algorithmic Model** section in the corresponding IBIS file.

```
|========================================================================
| This file contains the information for Algorithmic Model per
| IBIS Version 5.0 specification.
|
| Copy the content into the AMI Model Section in the corresponding ibis file.
|
| Note: This information is auto-generated by Agilent SystemVue software
|       based on the configuration of the PC workstation on which SystemVue runs.
|       If you use a compiler different from what is specified in the this file,
|       make sure to replace the compiler information when you transfer the
|       content here into the ibis file.
|
|========================================================================
[Algorithmic Model]
Executable Windows_VisualStudio9.0.30729_32 sv_ami_rx.dll sv_ami_rx.ami
Executable Windows_VisualStudio9.0.30729_64 sv_ami_rx_x64.dll sv_ami_rx.ami
[End Algorithmic Model]
```

Linear Time-Invariant System

If the AMI sub-network is an LTI (linear time-invariant) system, the AMI reserved parameter **Init_Returns_Impulse** is set to True, **GetWave_Exists** is set to False, and **Use_Init_Output** is set to True. In other words, the generated AMI_Init convolves the input channel impulse response with the impulse response of the AMI sub-network. The convolved response represents the modified channel impulse response including the AMI model behavior. The convolved response is returned in the first column of the impulse_matrix. Since the system is LTI, the convolution is equivalent to filtering the input channel impulse response samples using the AMI sub-network model. The same convolution process is also performed for each crosstalk channel, i.e., the rest of the columns in the input channel impulse response matrix are filtered using the AMI sub-network model and the convolved results are returned in place. AMI_GetWave function is not presented in the LTI case.

Use the "AMI_Tx_10" sub-network in the "IBIS AMI Modeling" workspace as an example. This workspace can be found in \Examples\Model Building\IBIS-AMI Modeling under SystemVue installation directory. The "AMI_Tx_10" sub-network is an LTI system, which is used in "SERDES_Tx_Rx_Schematic_10" schematic as an AMI transmitter and is associated with "AMI CPP 10 Gbps" code generator. The generated "sv_tx_10_ghz.ami" is shown below. Since the generated AMI_Init can handle arbitrary number of crosstalk channels, the Max_Init_Aggressors in the reserved parameters is set to the maximum possible value.

```
(sv_tx_10_ghz
(Reserved_Parameters
 (Init_Returns_Impulse (Usage Info) (Type Boolean) (Default True) (Description
"Init_Returns_Impulse True"))
 (GetWave_Exists (Usage Info) (Type Boolean) (Default False) (Description "GetWave_Exists False"))
 (Use_Init_Output (Usage Info) (Type Boolean) (Default True) (Description "Use_Init_Output True"))
 (Max_Init_Aggressors (Usage Info) (Type Integer) (Default 2147483646) (Description
"Max_Init_Aggressors 2147483646"))
)
(Model_Specific
 (Taps
  (0 (Usage In) (Type Tap) (Format Value 0) (Default 0) (Description "Taps 0"))
  (1 (Usage In) (Type Tap) (Format Value 1) (Default 1) (Description "Taps 1"))
  (2 (Usage In) (Type Tap) (Format Value 0) (Default 0) (Description "Taps 2"))
 )
 (Gain (Usage In) (Type Float) (Format Value 0) (Default 0) (Description "Gain"))
)
)
```

The generated "sv_tx_10_ghz_AMI.h" and "sv_tx_10_ghz_AMI.cpp" contains only AMI_init and AMI_Close. The header file is shown below.

```
/*
* sv_tx_10_ghz_AMI.h"
* Created by SystemVue C++ Code Generator
* Copyright © 2000-2011, Agilent Technologies, Inc.
*/
#pragma once
extern "C" __declspec(dllexport) long AMI_Init( double *impulse_matrix, long row_size, long aggress
ors, double sample_interval, double bit_time, char *AMI_parameters_in, char **AMI_parameters_out, v
oid **AMI_memory_handle, char **msg );
extern "C" __declspec(dllexport) long AMI_Close( void *AMI_memory );
```

The source file, "sv_tx_10_ghz_AMI.cpp", is presented in the following section of code.

```
/*
* sv_tx_10_ghz_AMI.cpp"
* Created by SystemVue C++ Code Generator
* Copyright © 2000-2011, Agilent Technologies, Inc.
*/
#include "sv_tx_10_ghz_AMI.h"
#include "sv_tx_10_ghz.h"
#include "SystemVue/Models/AMI/AmiParamParser.h"
#include <string>
#include <math.h>
class AMIContainer
{
public:
 sv_tx_10_ghz cSystemVueModel;
 std::string sMessage;
};
long AMI_Init( double *impulse_matrix, long row_size, long aggressors, double sample_interval, dou
ble bit_time, char *AMI_parameters_in, char **AMI_parameters_out, void **AMI_memory_handle, char
**msg )
{
 //Instantiate AMIContainer that manages AMI memory
 AMIContainer* pContainer = new AMIContainer;
 //SystemVue code generation model
 sv_tx_10_ghz* pSystemVueModel = &pContainer->cSystemVueModel;
 //AMI parameter parser
 AgilentEEsof::AMI::ParamParser parser( AMI_parameters_in );
 //samples per bit = bit_time / sample_interval, assume divisible
 int iSamplesPerBit = (int)(bit_time / sample_interval + 0.5);
 //set model parameters
 pSystemVueModel->SampleInterval = sample_interval;
 pSystemVueModel->SamplesPerBit = iSamplesPerBit;
 pSystemVueModel->Taps.Resize(3,1);
 if ( !parser.GetValue( pSystemVueModel->Taps(0), "sv_tx_10_ghz.Taps.0" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_tx_10_ghz, is unable to initialize the parameter
Taps.0 from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->Taps(1), "sv_tx_10_ghz.Taps.1" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_tx_10_ghz, is unable to initialize the parameter
Taps.1 from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->Taps(2), "sv_tx_10_ghz.Taps.2" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_tx_10_ghz, is unable to initialize the parameter
Taps.2 from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->Gain, "sv_tx_10_ghz.Gain" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_tx_10_ghz, is unable to initialize the parameter
Gain from the AMI_Init arguments.  The model will use the default setting. ";
 pSystemVueModel->Gain = pow( 10.0, pSystemVueModel->Gain / 20.0 ); //convert dB20 to MKS
 //allocate I/O buffers
 double* p_input_buffer = new double[ 1 ];
 pSystemVueModel->input.SetBuffer( p_input_buffer, 1, 1, 0 );
 double* p_output_buffer = new double[ 1 ];
 pSystemVueModel->output.SetBuffer( p_output_buffer, 1, 1, 0 );
 bool bStatus = true;
 //iterate through primary channel and aggressors
 long j;
 for ( j=0; j<=aggressors && bStatus; j++ )
 {
 //Setup
 if ( !pSystemVueModel->Setup() )
 {
  pContainer->sMessage += "Error occurred in sv_tx_10_ghz::Setup(). ";
  bStatus = false;
 }
 //Initialize
 if ( bStatus )
 {
  if ( !pSystemVueModel->Initialize() )
  {
   pContainer->sMessage += "Error occurred in sv_tx_10_ghz::Initialize(). ";
   bStatus = false;
  }
 }
 //modify impulse matrix
 if ( bStatus )
 {
  long i;
  for ( i=0; i<row_size; i++ )
  {
   pSystemVueModel->input[0] = impulse_matrix[ j*row_size + i  ];
   //Run
   if ( !pSystemVueModel->Run() )
   {
    pContainer->sMessage += "Error occurred in sv_tx_10_ghz::Run(). ";
    bStatus = false;
    break;
   }
   impulse_matrix[ j*row_size + i ] = pSystemVueModel->output[0];
  }
 }
 //Finalize
 if ( !pSystemVueModel->Finalize() )
 {
  pContainer->sMessage += "Error occurred in sv_tx_10_ghz::Finalize(). ";
```

```
  bStatus = false;
 }
 }
 //delete I/O buffers
 delete[] &pSystemVueModel->input[0];
 delete[] &pSystemVueModel->output[0];
 //set AMI_memory_handle
 *AMI_memory_handle = (void*)pContainer;
 //set msg
 if ( !pContainer->sMessage.empty() )
 *msg = (char*)pContainer->sMessage.c_str();
 return bStatus;
}
long AMI_Close( void *AMI_memory )
{
 //delete AMIContainer
 delete (AMIContainer*)AMI_memory;
 return 1;
}
```

As mentioned earlier, the actual execution routine for LTI system modifies the channel impulse response by filtering (processing) the input channel response samples using the generated sub-network model. See the section of codes commented as "//modify impulse matrix".

In "sv_tx_10_ghz.cpp", a container class, AMIContainer, is automatically created that contains the generated C++ sub-network model and provides additional messaging capabilities in AMI functions. AMI_Init uses AgilentEEsof::AMI::ParamParser to parse the given **AMI_parameters_in** string for setting the model specific parameters. If the parser is unable to parse or set a model specific parameter, a warning message will be generated and the particular parameter will be using the default value specified in the generated sub-network model.

For convenient parameter setting purpose, variable **iSamplesPerBit** is automatically computed in AMI_Init, which represents number of samples per bit (unit interval). The number of samples per bit is computed using bit_time / sample_interval, and is assumed to be an integer (divisible). This variable is commonly used in many AMI signal processing models (e.g., *BlindFFE* (algorithm)), and can be mapped to a sub-network parameter in AMI configuration tab.

SystemVue AMI code generator also provides a convenient feature for decibel unit conversion. If an AMI sub-network parameter is set to decibel-related unit using **Relative** or **Relative Power** unit category, the generated AMI_Init will automatically convert the parameter value back to MKS (absolute) unit. For example, the "Gain" parameter in "AMI_Tx_10" sub-network is set to dB unit using Relative unit category. As shown above, the code

```
pSystemVueModel->Gain = pow( 10.0, pSystemVueModel->Gain / 20.0 );
```

is inserted in AMI_Init to convert the value from dB to MKS.

The container class, parameter parser, messaging, samples-per-bit variable, and decibel unit conversion apply for both LTI and NLTV systems.

Non-Linear or Time-Variant System

If the AMI sub-network is an NLTV (non-linear or time-variant) system, the AMI reserved parameter **Init_Returns_Impulse** is set to False, **GetWave_Exists** is set to True, and **Use_Init_Output** is set to False. The input channel impulse response will not be modified. Instead, **AMI_GetWave** function is presented to filter input wave samples (**wave** argument) using the generated AMI sub-network model. The resulting wave samples are returned in-place using the same wave argument. NLTV AMI receivers can implement clock recovery algorithm and output recovered clock times using the **clock_times** argument. The clock_times is mapped to the **Clock Times** output port in AMI configuration tab.

Use the "AMI_Rx" sub-network in the "IBIS AMI Modeling" workspace as an example. This workspace can be found in \Examples\Model Building\IBIS-AMI Modeling under SystemVue installation directory. The "AMI_Rx" sub-network is a NLTV system, which is used in "SERDES_Tx_Rx_Schematic_10" schematic as an AMI receiver and is associated with "AMI CPP 10 Gbps" code generator. The generated "sv_ami_rx.ami" is shown below.

```
(sv_ami_rx
(Reserved_Parameters
 (Init_Returns_Impulse (Usage Info) (Type Boolean) (Default False) (Description
"Init_Returns_Impulse False"))
 (GetWave_Exists (Usage Info) (Type Boolean) (Default True) (Description "GetWave_Exists True"))
 (Use_Init_Output (Usage Info) (Type Boolean) (Default False) (Description "Use_Init_Output
False"))
 (Max_Init_Aggressors (Usage Info) (Type Integer) (Default 2147483646) (Description
"Max_Init_Aggressors 2147483646"))
)
(Model_Specific
 (LogicLevel (Usage In) (Type Float) (Format Value 5.0e-001) (Default 5.0e-001) (Description
"LogicLevel"))
 (NumberPrecursors (Usage In) (Type Integer) (Format Value 3) (Default 3) (Description
"NumberPrecursors"))
 (NumberPostcursors (Usage In) (Type Integer) (Format Value 3) (Default 3) (Description
"NumberPostcursors"))
 (DFETaps
  (0 (Usage In) (Type Tap) (Format Value 0) (Default 0) (Description "DFETaps 0"))
  (1 (Usage In) (Type Tap) (Format Value 0) (Default 0) (Description "DFETaps 1"))
  (2 (Usage In) (Type Tap) (Format Value 0) (Default 0) (Description "DFETaps 2"))
  (3 (Usage In) (Type Tap) (Format Value 0) (Default 0) (Description "DFETaps 3"))
  (4 (Usage In) (Type Tap) (Format Value 0) (Default 0) (Description "DFETaps 4"))
  (5 (Usage In) (Type Tap) (Format Value 0) (Default 0) (Description "DFETaps 5"))
 )
 (Alpha (Usage In) (Type Float) (Format Value 1.0e-003) (Default 1.0e-003) (Description "Alpha"))
)
)
```

The generated "sv_ami_rx_AMI.h" and "sv_ami_rx_AMI.cpp" are presented as follows.

```
/*
* sv_ami_rx_AMI.h"
* Created by SystemVue C++ Code Generator
* Copyright © 2000-2011, Agilent Technologies, Inc.
*/
#pragma once
extern "C" __declspec(dllexport) long AMI_Init( double *impulse_matrix, long row_size, long aggress
ors, double sample_interval, double bit_time, char *AMI_parameters_in, char **AMI_parameters_out, v
oid **AMI_memory_handle, char **msg );
extern "C" __declspec(dllexport) long AMI_GetWave( double *wave, long wave_size, double *
clock_times, char **AMI_parameters_out, void *AMI_memory );
extern "C" __declspec(dllexport) long AMI_Close( void *AMI_memory );
```

```
/*
* sv_ami_rx_AMI.cpp"
* Created by SystemVue C++ Code Generator
* Copyright © 2000-2011, Agilent Technologies, Inc.
*/
#include "sv_ami_rx_AMI.h"
#include "sv_ami_rx.h"
#include "SystemVue/Models/AMI/AmiParamParser.h"
#include <string>
#include <math.h>
class AMIContainer
{
public:
 sv_ami_rx cSystemVueModel;
 std::string sMessage;
 std::vector<double> vClockTimes;
 int iSamplesPerBit;
};
long AMI_Init( double *impulse_matrix, long row_size, long aggressors, double sample_interval, dou
ble bit_time, char *AMI_parameters_in, char **AMI_parameters_out, void **AMI_memory_handle, char
**msg )
{
 //Instantiate AMIContainer that manages AMI memory
```

```
 AMIContainer* pContainer = new AMIContainer;
 //SystemVue code generation model
 sv_ami_rx* pSystemVueModel = &pContainer->cSystemVueModel;
 //AMI parameter parser
 AgilentEEsof::AMI::ParamParser parser( AMI_parameters_in );
 //samples per bit = bit_time / sample_interval, assume divisible
 int iSamplesPerBit = (int)(bit_time / sample_interval + 0.5);
 pContainer->iSamplesPerBit = iSamplesPerBit;
 //set model parameters
 pSystemVueModel->SampleInterval = sample_interval;
 pSystemVueModel->SamplesPerBit = iSamplesPerBit;
 //pass only the impulse response of the primary channel
 pSystemVueModel->ImpulseMatrix.Resize( row_size, 1 );
 pSystemVueModel->ImpulseMatrix.CopyFrom( impulse_matrix, row_size );
 if ( !parser.GetValue( pSystemVueModel->LogicLevel, "sv_ami_rx.LogicLevel" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_ami_rx, is unable to initialize the parameter
LogicLevel from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->NumberPrecursors, "sv_ami_rx.NumberPrecursors" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_ami_rx, is unable to initialize the parameter
NumberPrecursors from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->NumberPostcursors, "sv_ami_rx.NumberPostcursors" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_ami_rx, is unable to initialize the parameter
NumberPostcursors from the AMI_Init arguments.  The model will use the default setting. ";
 pSystemVueModel->DFETaps.Resize(6,1);
 if ( !parser.GetValue( pSystemVueModel->DFETaps(0), "sv_ami_rx.DFETaps.0" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_ami_rx, is unable to initialize the parameter
DFETaps.0 from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->DFETaps(1), "sv_ami_rx.DFETaps.1" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_ami_rx, is unable to initialize the parameter
DFETaps.1 from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->DFETaps(2), "sv_ami_rx.DFETaps.2" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_ami_rx, is unable to initialize the parameter
DFETaps.2 from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->DFETaps(3), "sv_ami_rx.DFETaps.3" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_ami_rx, is unable to initialize the parameter
DFETaps.3 from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->DFETaps(4), "sv_ami_rx.DFETaps.4" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_ami_rx, is unable to initialize the parameter
DFETaps.4 from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->DFETaps(5), "sv_ami_rx.DFETaps.5" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_ami_rx, is unable to initialize the parameter
DFETaps.5 from the AMI_Init arguments.  The model will use the default setting. ";
 if ( !parser.GetValue( pSystemVueModel->Alpha, "sv_ami_rx.Alpha" ) )
 pContainer->sMessage += "The IBIS-AMI model, sv_ami_rx, is unable to initialize the parameter
Alpha from the AMI_Init arguments.  The model will use the default setting. ";
 pSystemVueModel->BitTime = bit_time;
 //allocate I/O buffers
 double* p_input_buffer = new double[ 1 ];
 pSystemVueModel->input.SetBuffer( p_input_buffer, 1, 1, 0 );
 double* p_clockTimes_buffer = new double[ 1 ];
 pSystemVueModel->clockTimes.SetBuffer( p_clockTimes_buffer, 1, 1, 0 );
 int* p_bit_buffer = new int[ 1 ];
 pSystemVueModel->bit.SetBuffer( p_bit_buffer, 1, 1, 0 );
 double* p_output_buffer = new double[ 1 ];
 pSystemVueModel->output.SetBuffer( p_output_buffer, 1, 1, 0 );
 bool bStatus = true;
 //Setup
 if ( !pSystemVueModel->Setup() )
 {
 pContainer->sMessage += "Error occurred in sv_ami_rx::Setup(). ";
 bStatus = false;
 }
 //Initialize
 if ( bStatus )
 {
 if ( !pSystemVueModel->Initialize() )
 {
  pContainer->sMessage += "Error occurred in sv_ami_rx::Initialize(). ";
  bStatus = false;
 }
 }
 //set AMI_memory_handle
 *AMI_memory_handle = (void*)pContainer;
 //set msg
 if ( !pContainer->sMessage.empty() )
 *msg = (char*)pContainer->sMessage.c_str();
 return bStatus;
}
long AMI_GetWave( double *wave, long wave_size, double *clock_times, char **AMI_parameters_out,
void *AMI_memory )
{
 AMIContainer* pContainer = (AMIContainer*)AMI_memory;
 sv_ami_rx* pSystemVueModel = &pContainer->cSystemVueModel;
 bool bStatus = true;
 long i, k = 0;
 //the upper bound for the possible number of clock times is wave_size + 1
 pContainer->vClockTimes.resize( wave_size + 1 );
 //filter wave and get clock times
 for ( i=0; i<wave_size; i++ )
 {
 //input wave sample
 pSystemVueModel->input[0] = wave[i];
 //Run
 if ( ! pSystemVueModel->Run() )
 {
  pContainer->sMessage += "Error occurred in sv_ami_rx::Run(). ";
  bStatus = false;
  break;
 }
 //output wave sample
 wave[i] = pSystemVueModel->output[0];
 //clock time
 if ( pSystemVueModel->clockTimes[0] >= 0 )
 {
  pContainer->vClockTimes[k++] = pSystemVueModel->clockTimes[0];
 }
 }
 //end clock_times
 pContainer->vClockTimes[k++] = -1;
 //From IBIS version 5.0 section 10, "The clock_time vector is allocated by the EDA platform and
is guaranteed to be greater than the number of clocks expected during the AMI_GetWave call."
 //However, the standard does not specify the size of clock_times array nor provide an argument in
AMI_GetWave for the size of clock_times array.
 //To prevent accessing clock_times array out of bounds, SystemVue AMI generator imposes a limit,
wave_size / samples_per_bit + 1 (including the last -1 element).
 //Users are free to modify this limit based on the EDA platform they use.
 long iClockTimesLimit = (long)floor((double)wave_size / (double)pContainer->iSamplesPerBit) + 1;
 if ( k <= iClockTimesLimit ) //k is the number of clock times
 {
 memcpy( clock_times, &pContainer->vClockTimes[0], sizeof(double)*k );
 }
 else
 {
 //if the number of clock times is larger than the limit, only return the last iClockTimesLimit
elements
 memcpy( clock_times, &pContainer->vClockTimes[ k - iClockTimesLimit ], sizeof(double)*iClockTimesL
imit );
 }
 return bStatus;
}
long AMI_Close( void *AMI_memory )
{
 AMIContainer* pContainer = (AMIContainer*)AMI_memory;
 sv_ami_rx* pSystemVueModel = &pContainer->cSystemVueModel;
 bool bStatus = true;
 //Finalize
 if ( !pSystemVueModel->Finalize() )
 {
 pContainer->sMessage += "Error occurred in sv_ami_rx::Finalize(). ";
 bStatus = false;
 }
 //delete I/O buffers
 delete[] &pSystemVueModel->input[0];
 delete[] &pSystemVueModel->clockTimes[0];
 delete[] &pSystemVueModel->bit[0];
 delete[] &pSystemVueModel->output[0];
 //delete AMIContainer
 delete pContainer;
 return bStatus;
}
```

In "sv_ami_rx_AMI.cpp", the input channel impulse response is not modified. However, signal processing models inside AMI sub-network can use the given channel impulse response to perform advanced operations, such as computing optimal FFE and DFE taps (See *FFE* (algorithm) and *DFE* (algorithm) models). The code

```
pSystemVueModel->ImpulseMatrix.CopyFrom( impulse_matrix, row_size );
```

passes the first column of the input **impulse_matrix**, i.e., the impulse response of the primary channel, into the AMI sub-network model.

> ⚠ The generated AMI_Init will only pass the first column, i.e., the primary channel impulse response, to the corresponding sub-network parameter.

Other AMI_Init input arguments are also passed into the AMI sub-network model for model setup and initialization.

```
pSystemVueModel->SampleInterval = sample_interval;
pSystemVueModel->SamplesPerBit = iSamplesPerBit;
pSystemVueModel->BitTime = bit_time;
```

AMI_GetWave filters the input **wave** samples using the Run() method of the generated AMI sub-network model. For each AMI_GetWave call, **wave_size** number of samples are processed, and the same number of output samples are returned using the same **wave** argument. To prevent data flow multirate mismatch, each execution (run) of the AMI sub-network model should consume one sample from the input port and produce one sample to each of the output port.

> ⚠ For AMI code generation, AMI sub-networks (including both LTI and NLTV systems) must preserve unit data flow production and consumption rates at the sub-network boundary for a complete sub-network execution. Please refer to *Introduction to Data Flow Simulation* (sim) for information about data flow production and consumption rates and scheduling.

Each invocation of pSystemVueModel->Run() consumes one input sample and produces one output wave sample and one clock time sample. However, on average, there is only one valid clock time for every **Samples Per Bit** number of samples. For samples that do not belong to valid clock time instances, the convention is to output negative numbers (see *ClockTimes* (algorithm) model).

```
//clock time
if ( pSystemVueModel->clockTimes[0] >= 0 )
{
 clock_times[k++] = pSystemVueModel->clockTimes[0];
}
```

> ⚠ AMI_GetWave will ignore negative clock time values and only return non-negative clock times.

Sharing Generated AMI Models with Others

The IBIS-AMI models that you generate can be copied onto other computers and shared with others. There are no dependencies on SystemVue or Visual Studio installations on the other computer. You must ship:

- The 32-bit and/or 64-bit DLLs compiled using Visual Studio. You should only ship the DLLs created using the Release Visual Studio configuration (as the debug libraries depend on the Visual Studio compiler being installed).
- The .ami file generated by SystemVue
- The .ibis file manually updated with the sv2011:Algorithmic Model definitions

Additionally, on the other computer you will have to install the MS Visual C++ Redistributable Packages:

- To support 32-bit AMI models, install: Microsoft Visual C++ 2008 SP1 Redistributable Package (x86)
- To support 64-bit AMI models, install: Microsoft Visual C++ 2008 SP1 Redistributable Package (x64)

For more details on redistributing MS Visual C++ run-time libraries, see: Redistributing Visual C++ Files

Importing Custom Intellectual Properties

Users can bring existing IPs (e.g., circuit designs, algorithms) from other environments into SystemVue through various ways. This section provides a general guideline to import custom IPs from HSPICE, Matlab, and C++.

HSPICE

For linear circuit in HSPICE, e.g., transmitter driver, users can obtain step response or impulse response from HSPICE and copy the response data into an equation page in SystemVue. For example, the following Mathlang code computes *StepResponse* and *StepResponseTimeStamps* from differential step response data, *step_n* and *step_p* obtained from HSPICE.

```
% Step response of negative level of differential circuit. Each row is a pair of time stamp and
step response value.
step_n = \[
0        1.1
2.5E-12 1.12
5.1E-12 1.13
...
\];
% Step response of positive level of differential circuit. Each row is a pair of time stamp and
step response value.
step_p = \[
0        0.9
2.5E-12 0.92
5.1E-12 0.93
...
\];
% Single-ended representation of step response to be used in simulation
StepResponse = step_p(:,2) - step_n(:,2);
% Time stamps for step response
StepResponseTimeStamps = step_p(:,1);
```

In SystemVue schematic, users can then use *TimeResponseFIR* (algorithm) model to represent such circuit. In this example, users can set the **ResponseType** option of *TimeResponseFIR* (algorithm) to *Step Response*; **Response** and **ResponseTimeStamps** parameters to *StepResponse* and *StepResponseTimeStamps* in the equation page, and set the **TimeStep** parameter to the simulation time step. For *TimeResponseFIR* (algorithm), the **ResponseTimeStamps** can have **non-evenly** spaced time stamps. *TimeResponseFIR* (algorithm) will interpolate and compute evenly spaced impulse response as FIR coefficients.

Matlab

Users can manually convert existing Matlab code into SystemVue C++ Model. Please refer to *Creating Custom C++ Model* (users) for detailed description.

Users can also use Matlab compiler to compile existing Matlab functions into C++ library, and then create SystemVue C++ models to wrap the library functions. Please refer to *Using Matlab Compiled Libraries in C++ Models* (users) for detailed description.

C++

Users can easily wrap existing C++ functions in SystemVue C++ models. Please refer to *Creating Custom C++ Model* (users) and *Using Third Party Library in C++ Models* (users) for detailed description.

Designs

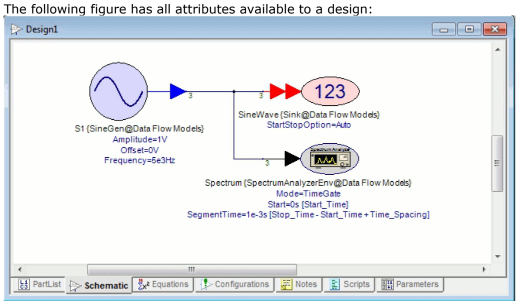
A design is an abstract term used to define a collection of related items that fully characterize a simulatable circuit. A design generally contains a schematic and parts list. However, notes, equations, scripts, and user defined parameters can also be added to any given design. The schematic is a visual representation of the parts being simulated and their connectivity with each other. The schematic is generally the heart of the design. Each tab at the bottom of the design window contains its characteristics that complement and influence the design. Many of these tabs and their added affects are optional. A design is the generic simulatable object.

A **schematic** is a design, but a design/schematic can also represent a schematic symbol, model, user model, sub-circuit, etc. Designs are often contained within designs (as a subcircuit). The most common names for a design are schematics, models, or parts. Throughout the documentation the term **Schematic** will be generally synonymous with **Design**.

Designs contain the following characteristics:

- Parts List (this is required)
- Schematic (optional but is generally the preferred method of entering part connectivity)
- Notes (optional)
- Equations (optional)
- Scripts (optional)
- Parameters (optional)
- Configurations (optional)

The following figure has all attributes available to a design:



Specific Types of Designs

A design is an abstract term used to define a collection of related items that fully characterize a simulatable circuit. Specific types of designs contain a specific set of these related items. All of the following items are specific types of designs:

- Schematic
- Schematic Symbol
- User Model

Contents

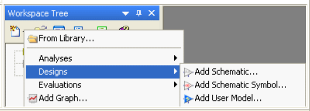
- *Creating a Design* (users)
- *Modifying a Design* (users)
- *Design Properties* (users)

Creating a Design

There are two different ways to to create a design in SystemVue. One is the by clicking on the **New Item button** ( ) on the Workspace Tree toolbar or by right clicking on a folder in the workspace tree.

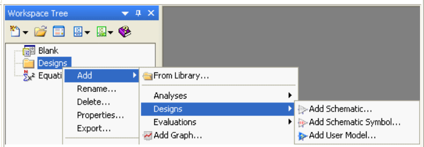
Method 1 - Clicking on the New Item Button

1. Click the New Item button ( ) on the Workspace Tree toolbar
2. Select the **'Designs >'** submenu
3. Now select the design of interest
4. The design will added under the folder that last selected in the workspace tree



Method 2 - Right Clicking on a Workspace Folder

1. Right click on a folder in the workspace tree to bring up the right click menu.
2. Select the 'Add >' submenu.
3. Select the 'Designs >' submenu
4. Now select the design of interest
5. The design will added under the folder that was initially right clicked

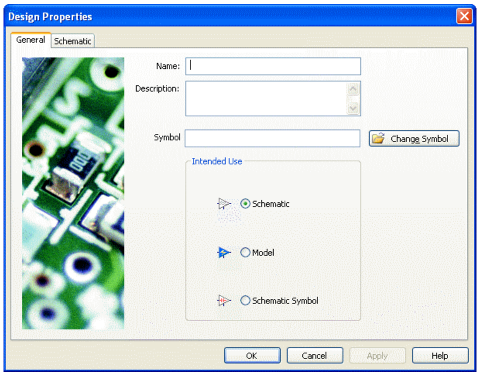


Note
If you create the new design in the wrong folder, simply drag it to the folder of interest.

Design Properties

General Tab

Use the General Properties tab page to change the general properties of a Design.



- **Name** - The name of the Design.
- **Description** - The Design description (optional).
- **Intended Use** - What kind of Design is it? This setting controls the Design's icon on the workspace tree and SystemVue' interpretation of how the design is intended to be used. (If it's a symbol, you can select it for a schematic part's symbol, if it's a model, you will be able to select it as a model, etc.)

Modifying a Design

The following attributes can be added to a design:

- Notes
- Equations
- Scripts
- Parameters
- Configurations

Here is a brief overview of these attributes:

Notes

A note can be added to help document different aspects of the design.

Equations

An equation block can be added to a design so that variables can be based on equations. These variables can be used for various design parameters. An equation block is generally the link between the parameters a user would see and the parameters used in models.

> **Note**
> These equations are local to the design. Other parts of your workspace cannot access the variables in this equation set.

For more information see *Equations* (users).

Scripts

Scripts control SystemVue operations. Add a script to your design to load files, save files, save data sets, and change object parameters.

> **Note**
> To run this script, copy the text and paste it into the Script Processor, then select Run.

For more information see *Scripts* (users).

Parameters

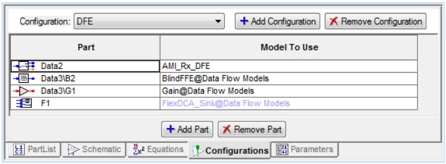
Parameters are added to a design when the implementation details are generally hidden from the user as is the case of a user model. When a design contains parameters this design can be used as a user model and these parameters will be exposed as model parameters in the part that uses this design as its model.

For more information see *User Defined Parameters* (users).

Configurations

A configuration tells an Analysis or a Code Generator to use specific model for a part which is already included in the manage model list of that part. You can add multiple configurations in a Configurations tab and select one specific configuration inside the Analysis or the Code Generator.

In the following example, the configuration DFE will use specific model for the 4 parts which are added, including 2 inside sub networks (Data3\B2 and Data3\G1). For all the other parts, the currently selected model will be used for this configuration



- **Configuration** – A configuration is a set of Part/Model pair which will be used during simulation and code generation if that configuration is selected in the corresponding Analysis or Code Generator. You only need to add those parts in a configuration for which you would like to use a model which is not currently set as the active model for the part in the schematic. All the parts in a design which are not added to a configuration will use their current active model as shown in the schematic.

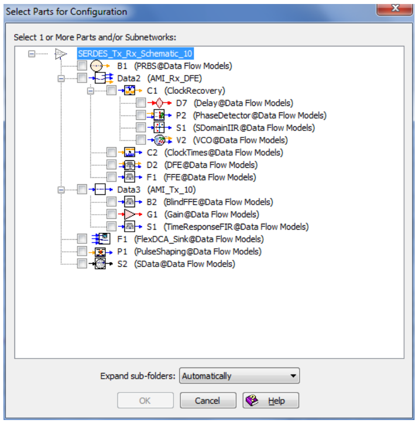
  > **Note**
  > Only the configuration from Model Manager of the top level designs are available for selection in Analysis and Code Generators.

- **Adding a Configuration** – When you add the Model Manager tab for the first time, it will prompt you to add the name of the first configuration and add it automatically. If you need to add another configuration for the design use **"+ Add Configuration"** button.

- **Removing Configuration(s)** – To remove configuration(s) from Model Manager tab click on the button **"Remove Configuration"**, it will open a dialog box, as shown below, select the configuration(s) to be deleted and click on **"Delete Selected"** button.

- **Adding Parts to Configuration"** – Select the configuration, for which you would like to add a part and click **"+AddPart"** button to open the following dialog box

> **Note**
> It is advisable that you should not select a part which is in the hierarchy of a part that you have already selected. If you do that and change the model for top level part, then the configuration will not be valid.



- **Removing a Part From Configuration"** – Select the part to be removed and click **"Remove Part"**.

- **Selecting a Model To Use** – Click a part's **Model To Use** field to select models from the manage model list for that part. When a part's **Model To Use** field is blue, it means that there is only 1 available model for that part. Bring up the part in Part Properties and use the Model Manager to add additional models to the part.

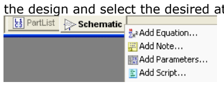
Using a Configuration

Configurations to a top level design can be used inside *Data Flow Analysis* (sim) (please read *Setting up the Data Flow Analysis* (sim)), and/or inside C++ Code Generation Analysis (Please read Adding a C++ Code Generator Analysis) to select a particular configuration during simulation or code generation.

This feature is extremely useful and powerful when you have different flavor of an algorithm implemented with different models added to manage model list of a part. For example, you may have several implementation of FIR as C++ model, as VHDL/Verilog (using HDL model in SystemVue), as a Math Language model, and a built in SystemVue FIR. By using this feature you can setup configurations, and then in your Data Flow Analysis or a Code Generator, you can select a particular configuration without manually going to each part and manually switching the model.

Adding a Design Attribute

To add one of these attributes to a design right click on one of the tabs at the bottom of the design and select the desired attribute.



Deleting a Design Attribute

A tab can also be deleted from a design my right clicking on it and selecting the 'Delete' menu entry.

Filter Designer

SystemVue **Filter Designer** is a filter design tool that helps users to design **digital IIR** (infinite impulse response) and **FIR** (finite impulse response) filters based on the specified frequency response, design method, and other relevant parameters.

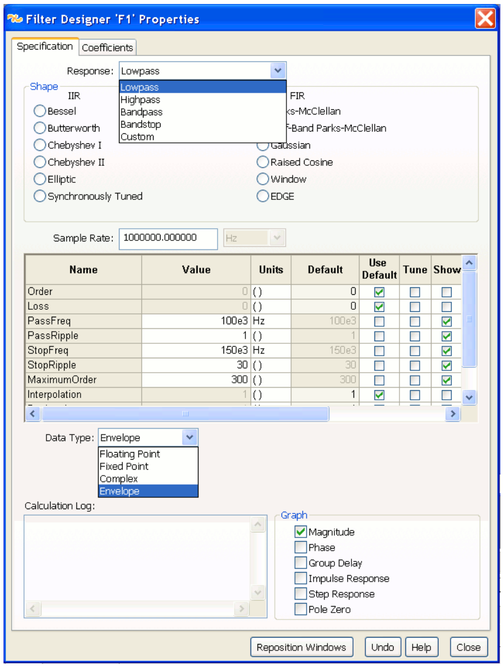
SystemVue integrates Filter Designer with the *Filter Part* (algorithm). To launch the Filter Designer, place a **Filter part** on a schematic and double click the Filter part. For the associated filter models, you can also right click the part and select "Filter Designer...".

> Please refer to *Filter Part* (algorithm) for further details. Especially refer to *Filter Designer and Filter Part* (algorithm) about the integration with the Filter Part and the associated filter models.

The Filter Designer is implemented as a "live" dialog box — as specifications are changed, the plots and coefficients are updated automatically. With this feature, users can easily experiment with different design methods and parameters, and verify the responses as well as review the coefficients in almost real time.

> Certain filter specifications may require large filter order and cost noticeable amount of time to design the filter and to compute the responses. For example, Sample Rate is too large comparing to the frequency specifications, or the specified transition band is too small to be accomplished, etc. Under these conditions, the Filter Designer may be busy in designing the filter and computing the responses and may not respond to user's action.

Filter Specification Window



- **Response** - Lowpass, Highpass, Bandpass, Bandstop, or Custom (user specified coefficients/responses).
- **Shape (Design Method)** - IIR design methods include Bessel, Butterworth, Chebyshev I, Chebyshev II, Elliptic, Synchronously Tuned, S-domain poles-zeros, etc. FIR design methods include Parks-McClellan, Raised Cosine, Gaussian, Window, EDGE pulse shaping, frequency response specification, etc. The available design methods may vary depending on different response selection. Please refer to *IIR Filter Design* (users) and *FIR Filter Design* (users) for introduction.
- **Sample Rate** - The sampling rate that is used to design the digital filter. All the frequency specifications are relative to the specified Sample Rate.

  > For **Envelope** data type, this Sample Rate is used in Filter Designer ONLY. During simulation, the filter model will re-design the filter based on the resolved input sampling rate.

  > For **Floating Point**, **Fixed Point**, and **Complex** data types, the sampling rate of the incoming signal, no matter what the value is, is always mapped to 2π of the numeric filter.

  > If the Sample Rate is too small to represent a bandpass or a bandstop filter, the Filter Designer will then try to design the filter for **analytic signal** (see *Bandpass and Bandstop Filtering for Analytic Signals* (sim)) and inform the user in the calculation log. This additional design process is for Filter Designer ONLY. During simulation, the filter model will re-design the filter based on the actual input signal (real or analytic).

- **Parameters** - The parameter grid is filled with the specific parameters of the selected filter. After entering a parameter value in the Value column, press Enter to accept the value and move to the next row.

  > Please refer to *individual filter model documentation* (algorithm) for parameter details.

- **Data Type** - Specify the data type of the simulation model. After closing the filter designer, a particular simulation model will be instantiated based on the design specification and data type.
  - **Floating Point** - If the filter specification is IIR, either *IIR* (algorithm) or *BiquadCascade* (algorithm) will be instantiated depending on whether the IIR coefficients can fit into the cascade-biquad structure. If the filter specification is FIR, *FIR* (algorithm) will be instantiated as the simulation model. The resulting coefficients (taps) of the current filter design will be used as IIR or FIR coefficients (taps).
  - **Fixed Point** - If the filter specification is FIR, *FIR_Fxp* (hardware) will be instantiated as the simulation model. The resulting coefficients of the current filter design will be used as the coefficients of *FIR_Fxp* (hardware) before quantization. See Fixed Point Filter Design for additional fixed point specification.
  - **Complex** - Filter designer will instantiate *IIR_Cx* (algorithm) if the filter specification is IIR or *FIR_Cx* (algorithm) if the filter specification is FIR. The resulting coefficients (taps) of the current filter design, in most cases, real-valued coefficients, will be used directly as the coefficients (taps) of *IIR_Cx* (algorithm) (or *FIR_Cx* (algorithm)) even though the input and output signals are in complex format.
  - **Envelope** - Filter designer will instantiate a particular black pin filter model, based on the response and shape selection, for filtering *Envelope Signal* (sim). See *Associated Filter Models* (algorithm) for a list of the associated envelope filter models. See also *Filtering Envelope Signal* (sim) for technical details about how SystemVue filter models filter real and analytic signal.

- **Graph** - Check the boxes to display the designated plots (magnitude response, phase response, group delay, impulse response, step response, and pole-zero plot).
- **Reposition Windows** - Click to restore the enabled graph windows to their default arrangement.

- **Undo** - Undoes all setting changes (restores settings to original values when the Filter Designer popped up). This is like a Cancel operation, but the dialog box remains open.
- **Help** - Displays THIS help topic.
- **Close** - Closes the Filter Designer window, instantiates the proper filter model (as well as the proper symbol) based on the design specification and data type under the Filter part, and retains the current parameter values for the filter model.

Fixed Point Filter Design

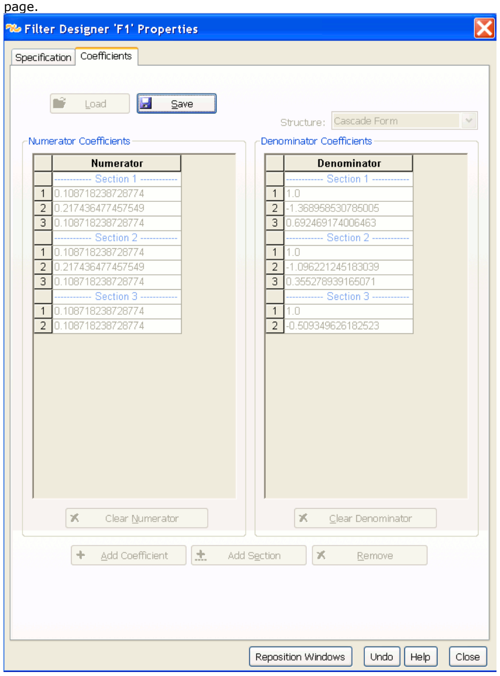
When the filter specification is FIR and the data type is **Fixed Point**, additional fixed point properties will be displayed in the filter designer.

- **Word Length** - Specify the word length of fixed point filter taps. The specified value will be copied to the **WeightsWordlength** parameter of *FIR_Fxp* (hardware).
- **Integer Word Length** - Specify the integer word length of fixed point filter taps. The specified value will be copied to the **WeightsIntegerWordlength** parameter of *FIR_Fxp* (hardware).
- **Quantization** - Specify the quantization mode of fixed point filter taps. The selection will be copied to the **WeightsQuantization** parameter of *FIR_Fxp* (hardware).
  - RND - Rounding to Plus infinity.
  - RND_ZERO - Rounding to Zero.
  - RND_MIN_INF - Rounding to Minus infinity.
  - RND_INF - Rounding to infinity.
  - RND_CONV - Convergent rounding.
  - TRN - Truncation.
  - TRN_ZERO - Truncation to zero.
- **Overflow** - Specify the overflow mode of the fixed point filter taps. The specified value will be copied to the **WeightsOverflow** parameter of *FIR_Fxp* (hardware).
  - SAT - Saturation.
  - SAT_ZERO - Saturation to Zero.
  - SAT_SYM - Symmetrical saturation.
  - WRAP - Wrap-around.
  - WRAP_SM - Sign magnitude wrap-around.

Users can modify the fixed point properties and see the quantized responses in the fixed point response plot.

Coefficients Display Window

To see the filter coefficients, click on the "Coefficients" tab to display the Coefficients page.



- **Save** - Saves the filter coefficients file; the file format is compatible with SystemVue Classic.

Most of the other settings are only available when using a Custom Z-Domain filter (CustomIIR or CustomFIR / Taps).

- **Load** - Loads a filter coefficients file; the file format is compatible with SystemVue Classic.
- **Structure** - Determines the IIR filter structure which can be either cascade form or parallel form. This setting is ignored for FIR filters.
- **Coefficients** - The coefficients grid is filled with the filter's coefficients. After entering a setting, press Enter to accept the value and move to the next row. IIR filters are implemented as sections.
- **Clear Numerator / Denominator** - Sets the number of coefficients to 1 and sets its value to "1.0".
- **Add Coefficient** - Inserts a new coefficient after the current selection and sets its value to "1.0".
- **Add Section** - Inserts a new section after the currently selected section. The new section will have one coefficient set to "1.0".
- **Remove** - Removes the currently selected coefficient or section. The section is removed from both numerator and denominator.
- **Reposition Windows** - Click to restore the enabled graph windows to their default arrangement.
- **Undo** - Undoes all setting changes (restores settings to original values when the Filter Designer popped up). This is like a Cancel operation, but the dialog box remains open.
- **Help** - Displays THIS help topic.
- **Close** - Closes the window, retaining the current settings.

For FIR filters, the coefficients shown in the "Coefficients" tab do not include decimation process.

Response Plots

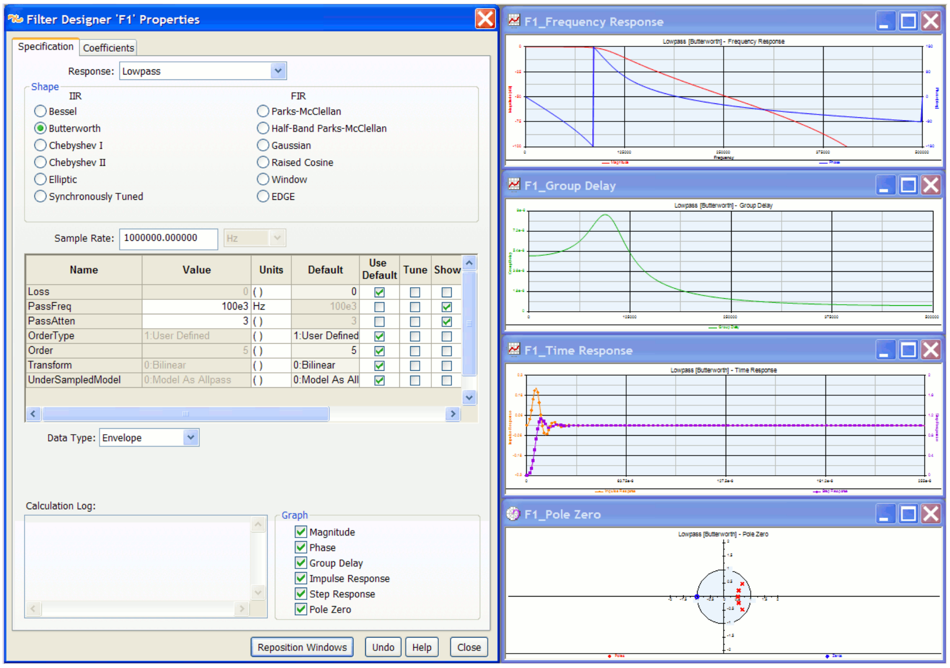
In the right-lower corner of the specification window, users can choose to display:

- **Magnitude Response** - Magnitude response of the filter is displayed in the **Frequency Response** window. Frequency (X axis) is from 0 to half the sampling rate (Sample Rate / 2) in Hz. Magnitude (left Y axis) is in dB by default and is switchable to absolute unit.
- **Phase Response** - Phase response of the filter is displayed in the **Frequency Response** window. Frequency (X axis) is from 0 to half the sampling rate (Sample Rate / 2)in Hz. Phase (right Y axis) is in radius.
- **Group Delay** - Group delay of the filter is displayed in the **Group Delay** window. Frequency (X axis) is from 0 to half the sampling rate (Sample Rate / 2) in Hz. Group delay is in second. If the magnitude at certain frequency is too small to give good fidelity in group delay computation, the group delay at that particular frequency is 0 by default.

When the Filter Designer designs a bandpass or bandstop filter for **analytic signal** (see **Sample Rate** in Filter Specification Window and also see *Bandpass and Bandstop Filtering for Analytic Signals* (sim)), the frequency axis is adjusted to the range from FCenter - Sample Rate / 2 to FCenter _ Sample Rate / 2.

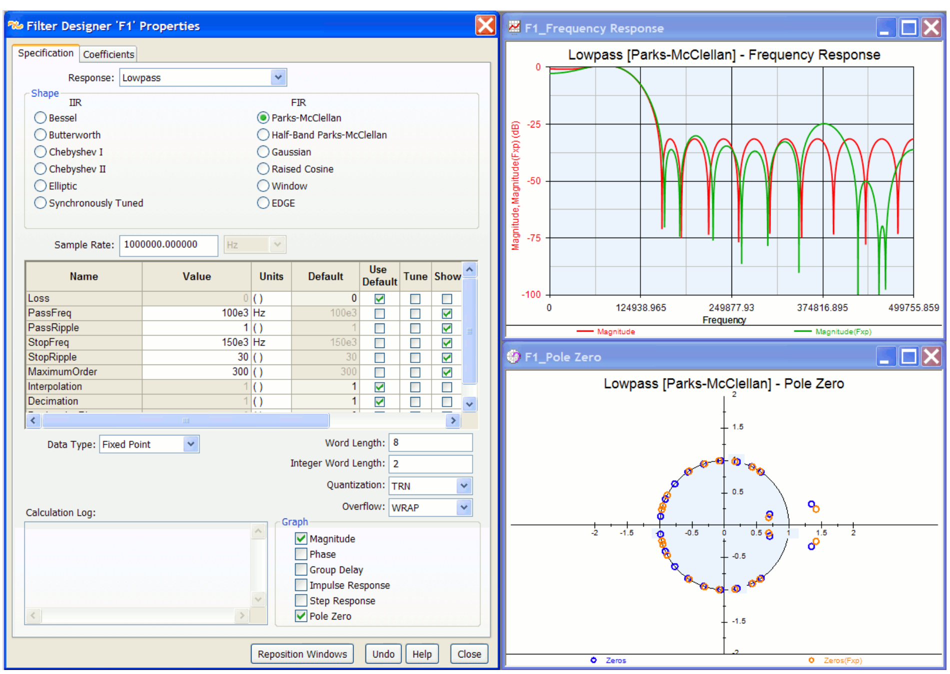
- **Impulse Response** - Impulse response of the filter is displayed in the **Time Response** window. Time (X axis) is in second. Time step is 1 / sampling rate. Impulse response is in left Y axis.
- **Step Response** - Step response of the filter is displayed in the **Time Response** window. Time (X axis) is in second. Time step is 1 / sampling rate. Step response is in right Y axis.
- **Pole Zero Plot** - Poles and zeros of the IIR filter or zeros of the FIR filter are displayed in the Pole-Zero plot.

  It is users' responsibility to make sure the Sample Rate and the Interpolation and Decimation factors (for FIR only) are properly set. The frequency response and group delay plots cannot display imaging effects and may not display aliasing effects properly.



Response Plots for Fixed Point Data Type

When **Fixed Point** data type is selected, there will be two traces in each plot, one for floating point and the other for fixed point based on the specified fixed point properties. Refer to ***FIR_Fxp* (hardware)** for more details on Fixed Point FIR.



FIR Filter Design

The following sections introduce the technical basics and various design methods for FIR filters. Please refer to the references and other textbooks or documents for detailed description.

- *Causal, Linear Phase FIR Filter Basic* (users)
- *Window Method* (users)
- *Parks-McClellan Method* (users)
- *Gaussian Filter* (users)
- *Raised Cosine Filter* (users)
- *EDGE Pulse Shaping Filter* (users)
- *Custom FIR Design* (users)
- *Multirate Polyphase FIR Filter Implementation* (users)

References

1. S. Haykin, *Communication Systems*, 4th ed. John Wiley and Sons, Inc, 2000.
2. A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*, 2nd ed. Prentice Hall, 1999.
3. B. Sklar, *Digital Communications: Fundamentals and Applications*. Prentice Hall, 1988.
4. P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice Hall, 1993.

Causal, Linear Phase FIR Filter Basic

According to [2], for a causal and linear phase FIR system, if the impulse response is symmetric, i.e.,

$$h[n] = \begin{cases} h[M-n], & 0 \le n \le M \ , \\ 0, & \text{otherwise,} \end{cases}$$

then the frequency response is
$H(e^{j\omega}) = R_e(e^{j\omega})e^{-j\omega M/2}$,
where $R_e(e^{j\omega})$ is a real, even, and periodic function of ω.

On the other hand, for a causal and linear phase FIR system, if the impulse response is antisymmetric, i.e.,

$$h[n] = \begin{cases} -h[M-n], & 0 \le n \le M \ , \\ 0, & \text{otherwise,} \end{cases}$$

then the frequency response is
$H(e^{j\omega}) = jR_o(e^{j\omega})e^{-j\omega M/2} = R_o(e^{j\omega})e^{-j\omega M/2 + j\pi/2}$,
where $R_o(e^{j\omega})$ is a real, odd, and periodic function of ω.

Four types of causal, linear-phase FIR filters are defined in [2], and the four types are listed in the following table.

Type	Symmetry	Order M	Frequency Response	$\omega = 0$	$\omega = \pi$
I	Symmetric	Even	$H(e^{j\omega}) = e^{-j\omega M/2}\left(\sum_{k=0}^{M/2} a[k]\cos(\omega k)\right)$ $a[0] = h[M/2]; a[k] = 2h[M/2-k], k = 1, 2, \ldots, M/2$	No restriction	No restriction
II	Symmetric	Odd	$H(e^{j\omega}) = e^{-j\omega M/2}\left(\sum_{k=1}^{(M+1)/2} b[k]\cos(\omega(k-1/2))\right)$ $b[k] = 2h[(M+1)/2-k], k = 1, 2, \ldots, (M+1)/2$	No restriction	$H(e^{j\pi}) = 0$
III	Antisymmetric	Even	$H(e^{j\omega}) = je^{-j\omega M/2}\left(\sum_{k=1}^{M/2} c[k]\sin(\omega k)\right)$ $c[k] = 2h[M/2-k], k = 1, 2, \ldots, M/2$	$H(e^{j0}) = 0$	$H(e^{j\pi}) = 0$
IV	Antisymmetric	Odd	$H(e^{j\omega}) = je^{-j\omega M/2}\left(\sum_{k=1}^{(M+1)/2} d[k]\sin(\omega(k-1/2))\right)$ $d[k] = 2h[(M+1)/2-k], k = 1, 2, \ldots, (M+1)/2$	$H(e^{j0}) = 0$	No restriction

According to the table, Type II is not suitable for highpass nor bandstop filters; Type III is only acceptable for bandpass filter; and Type IV is not suitable for lowpass nor bandstop filters.

Custom FIR Design

In custom FIR design, users specify the desired frequency response of the FIR filter. In general, the frequency response is specified in terms of

- a finite set of frequency points (in Hz), $\{f_0, f_1, \ldots, f_N\}$, where $0 \le f_0 < f_1 < \ldots < f_N$
- a magnitude response (in dB) $\{|H(f_0)|, |H(f_1)|, \ldots, |H(f_N)|\}$
- a phase response (in degrees) $\{\angle H(f_0), \angle H(f_1), \ldots, \angle H(f_N)\}$.

There are several methods to compute FIR coefficients based on the given frequency response. The SystemVue *CustomFIR* (algorithm) and *SData* (algorithm) models use a method that involves the FFT.

The data provided is first interpolated as needed to obtain a power of two number of data points that are evenly spaced in frequency.

- When the input signal is real, the first frequency point must be at 0 Hz and the data is interpolated in the range [0, (*Sample Rate*) / 2].
- When the input signal is a complex envelope one with non-zero characterization frequency, fc, the first frequency point can be greater than zero and the data is interpolated in the range [fc - (*Sample Rate*) / 2, fc + (*Sample Rate*) / 2].

Any frequency response data supplied that is outside the above ranges is not used.

If the supplied frequency response data does not extend to the limits of the ranges defined above, data extrapolation is performed to achieve data to these limits. The extrapolation method used, as specified in the model *ExtrapolationOption* parameter, can be

- Constant: the value at the lowest/highest frequency specified is held constant until the limits of the ranges defined above are reached.
- versus freq: linear extrapolation is performed.

An additional extrapolation roll-off, as specified in the model *ExtrapolationRollOff* parameter in terms of dB/octave, can also be applied.

From the interpolated frequency data, the inverse FFT is applied to obtain a set of FIR coefficients. For a complex envelope input signal, the interpolated frequency data is decomposed into I and Q frequency responses, which are used to generate the I and Q FIR filters that will independently filter the complex envelope input signal I and Q envelopes.

When the magnitude tolerance, as specified in the model *MagTolerance* parameter (in dB), is greater than zero, the FIR coefficients obtained from the inverse FFT are further processed to possibly reduce the total number of coefficients needed to represent the frequency response data as an FIR filter. This is done by successively reducing the number of FIR coefficients, observing the resultant frequency response, and stopping the coefficient number reduction when the specified magnitude tolerance is reached. Oftentimes, a 1 dB magnitude tolerance can result in a sizable reduction in the number of FIR coefficients. During this process, the magnitude tolerance is calculated only over the tolerance frequency range [*LowerFitFreq*, *UpperFitFreq*], as specified in corresponding model parameters.

EDGE Pulse Shaping Filter

EDGE pulse shaping filter is the modulation pulse shaping filter; it is used to control the power of the spectrum outband and decrease the peak-to-average ratio. The impulse response of this filter is $C_0(t)$, which is the main component in the Laurent expansion of the GMSK modulation. In his paper, Laurent introduces a method to express any constant-amplitude binary phase modulation as a sum of a finite number of time-limited amplitude-modulated pulses (AMP decomposition). Using this method in GMSK, which is a constant-amplitude phase modulation, the GMSK signals can be transformed into the sum of $C_0(t), C_1(t), \ldots, C_M(t)$, where M is derived from the length of the impulse response of the Gaussian filter. And, compared to $C_0(t)$, other components $C_1(t), \ldots, C_M(t)$ are all negligible.

Given the symbol rate $1/T$ hz, the impulse response of the EDGE pulse shaping filter $C_0(t)$ is defined as follows.

$$C_0(t) = \begin{cases} \prod_{i=0}^{3} S_i(t), & 0 \le t \le 5T, \\ 0, & \text{otherwise}, \end{cases}$$

where

$$S_i(t) = S_0(t + iT)$$

and

$$S_0(t) = \begin{cases} \sin\left(\pi \int_{-\infty}^{t} g(\tau) d\tau\right), & 0 \le t \le 4T, \\ \sin\left(\pi/2 - \pi \int_{-\infty}^{t-4T} g(\tau) d\tau\right), & 4T \le t \le 8T, \\ 0, & \text{otherwise}, \end{cases}$$

where $g(t)$ is the rectangular pulse response of the Gaussian filter in GMSK modulation.

$$g(t) = \frac{1}{2T}\left(Q\left(2\pi \times 0.3 \times \frac{t - 5T/2}{T\sqrt{\ln 2}}\right) - Q\left(2\pi \times 0.3 \times \frac{t - 3T/2}{T\sqrt{\ln 2}}\right)\right)$$

and

$$Q(t) = \frac{1}{\sqrt{2\pi}} \int_{t}^{\infty} e^{-\tau^2/2} d\tau.$$

EDGE References

- P. A. Laurent, "Exact and Approximate Construction of Digital Phase Modulations by Superposition of Amplitude Modulated Pulses (AMP)," *IEEE Trans. Commun.*, vol. COM-34, NO. 2, pp. 150-160, Feb. 1986.
- P. Qinhua, G. Yong and L. Weidong, "Synchronization Design Theory of Demodulation for Digital Land Mobile Radio System," *Journal of Beijing University of Posts and Telecommunications*, Vol. 18, No. 2, pp. 14-21, Jun. 1995.
- ETSI Tdoc SMG2 WPB 108/98, Ericsson, EDGE Evaluation of 8-PSK.

Gaussian Filter

Gaussian filters have the special pulse filtering property, that is, they provide the fastest pulse rise time with no overshoot or ringing in time domain.

Lowpass Gaussian Filter, see [*1* (users)], is defined as

$$H(f) = \exp\left(-\frac{\ln 2}{2}\left(\frac{f}{f_{3db}}\right)^2\right),$$

and the corresponding impulse response is

$$h(t) = \left(\frac{2\pi}{\ln 2}\right)^{1/2} f_{3db} \exp\left(-\frac{2\pi^2}{\ln 2} f_{3db}^2 t^2\right).$$

Here, $H(f_{3db}) = 1/\sqrt{2}$.

The discrete-time FIR lowpass Gaussian filter is designed by introducing delay d in $h(t)$ and then sampling the delayed version starting from $t = 0$ up to filter order M.

$$h[n] = \begin{cases} h(n\Delta t - d), & 0 \le n \le M, \\ 0, & \text{otherwise}, \end{cases}$$

where Δt is the sampling period.

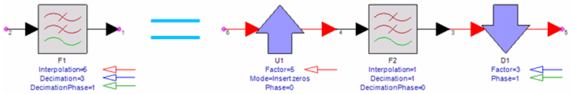
In SystemVue, bandpass Gaussian filter with 3dB bandwidth w_{3db} and center frequency f_c is defined as

$$h_{bp}(t) = h(t) \times 2cos(2\pi f_c t),$$

where $h(t)$ is the lowpass Gaussian filter defined above with $f_{3db} = w_{3db}/2$.

Multirate Polyphase FIR Filter Implementation

Most of the SystemVue FIR filter blocks are integrated with multirate (rational sampling rate change) capability. Users can specify *Interpolation* factor, *Decimation* factor, and *DecimationPhase* for the desired multirate characteristics. By default (*Interpolation* = 1, *Decimation* = 1, and *DecimationPhase* = 0), the filter blocks do not perform any rate change. When the *Decimation* factor is > 1, the FIR filter behaves exactly as if it were in default mode and were followed by a *DownSample* (algorithm) block with *Factor* parameter equal to the *Decimation* factor of the filter and *Phase* parameter equal to the *DecimationPhase* of the filter. Similarly, when the *Interpolation* factor is > 1, the filter behaves as if it were in default mode and were preceded by an *UpSample* (algorithm) block with *Factor* parameter equal to the *Interpolation* factor of the filter, *Mode* parameter is "Insert zeros" (zero insertion), and *Phase* parameter equal to 0 (interpolation phase is 0). The following figure illustrate the equivalence of SystemVue multirate FIR filters.



A subset of multirate filter models provide additional *InterpolationScaling* parameter to specify whether the output signal should be multiplied by the *Interpolation* value when *Interpolation* factor is larger than 1. The purpose of *InterpolationScaling* is to adjust the magnitude of the output signal to compensate the zero insertion during up-sampling. By default, *InterpolationScaling* is YES.

> The equivalence in the above figure holds when *InterpolationScaling* is NO.

The benefit of multirate polyphase filters is that the multirate implementation integrated inside the filter models is much more efficient than it would be using *UpSample* (algorithm) and *DownSample* (algorithm). A **polyphase** structure is used internally, avoiding unnecessary use of memory and unnecessary multiplication of zeros. Arbitrary sample-rate conversions by rational factors can be accomplished this way.

> It is users' responsibility to make sure the decimation and interpolation factors do not cause aliasing and imaging effect. The *Filter Designer* (users) frequency response plot cannot display imaging effect and may not properly display aliasing effect.

Parks-McClellan Method

The Parks-McClellan design method uses the Remez exchange algorithm to design linear phase FIR filters such that a filter has minimum weighted Chebyshev error in approximating a desired ideal frequency response. For further details, please refer to Chapter 7.4.3 The Parks-McClellan Algorithm in *Discrete-Time Signal Processing*, 2nd ed. [*2* (users)].

The Parks-McClellan design method in SystemVue uses a modified version of the remez program from Jake Janovetz under GNU Library General Public License. The source of the modified remez program and the GNU Library General Public License are located in PublicSource\remez under the SystemVue installation directory.

Raised Cosine Filter

Raised-cosine filters are used for shaping pulses for transmission through digital channels to prevent intersymbol interference (ISI). The background for intersymbol interference and raised cosine filter can be found in [1], [3], and other communication textbooks. The following discussion is based on [3].

The minimum system bandwidth to detect $\frac{1}{T}$ symbols/sec without ISI is $F_0 = \frac{1}{2T}$ hz. However, in practical, we need to provide some "excess bandwidth" beyond the theoretical minimum. One frequently used system transfer function is the **raised cosine filter**.

The lowpass raised cosine filter has transfer function

$$H_{rc}(f) = \begin{cases} 1, & |f| < F_0(1-R), \\ \left(\cos\left(\frac{2\pi f - 2\pi F_0(1-R)}{8F_0R}\right)\right)^2, & F_0(1-R) \le |f| \le F_0(1+R), \\ 0, & |f| > F_0(1+R), \end{cases}$$

where R is the **roll-off factor** between 0 and 1, and F_0R is the **excess bandwidth** over the ideal Nyquist bandwidth $F_0 = \frac{1}{2T}$.
The impulse response of the raised cosine filter is
$$h_{rc}(t) = \frac{\sin(\pi t/T)}{\pi t/T} \cdot \frac{\cos(\pi Rt/T)}{1-(2Rt/T)^2}.$$

The transfer function of the **square root raised cosine filter** or **root raised cosine filter** is defined as
$$H_{rrc}(f) = (H_{rc}(f))^{1/2}.$$
The corresponding impulse response is
$$h_{rrc}(t) = \frac{4R}{\pi\sqrt{T}} \cdot \frac{\cos((1+R)\pi t/T) + \frac{\sin((1-R)\pi t/T)}{4Rt/T}}{1-(4Rt/T)^2}.$$

The above raised cosine filter and root raised cosine filter are defined in continuous-time domain, and the impulse responses are not causal. The discrete-time raised cosine and root raised cosine FIR filters are obtained by introducing delay d in $h_{rc}(t)$ and $h_{rrc}(t)$ and then sampling the delayed versions starting from $t = 0$ up to the filter order M.

$$h_{rc}[n] = \begin{cases} h_{rc}(n\Delta t - d), & 0 \le n \le M \\ 0, & \text{otherwise.} \end{cases} \quad h_{rrc}[n] = \begin{cases} h_{rrc}(n\Delta t - d), & 0 \le n \le M \\ 0, & \text{otherwise.} \end{cases}$$

The transfer function of the raised cosine filter with pulse equalization is
$$H_{rce}(f) = \frac{2\pi f/(4F)}{\sin(2\pi f/(4F))} H_{rc}(f),$$
and the transfer function of the root raised cosine filter with pulse equalization is
$$H_{rrce}(f) = \frac{2\pi f/(4F)}{\sin(2\pi f/(4F))} H_{rrc}(f).$$

In SystemVue, the impulse response of the discrete-time pulse equalization raised cosine $h_{rce}[n]$ (or root raised cosine $h_{rrce}[n]$) filter is computed by first sampling $H_{rce}(f)$ (or $H_{rrce}(f)$) in equally spaced frequency points, next performing inverse discrete Fourier transform (IDFT), and then take the real parts.

Window functions can be applied to raised cosine and root raised cosine filters to smooth the possible discontinuities at the both ends of the impulse response.

In SystemVue, bandpass raised cosine filter with center frequency f_c is defined as
$$h_{bp}(t) = h_{lp}(t) \times 2cos(2\pi f_c t),$$
where $h_{lp}(t)$ is a particular lowpass raised cosine filter defined above.

Window Method

Designing FIR filters using **window method** generally begins with an ideal desired frequency response $H_d(e^{j\omega})$. After that, the ideal impulse response $h_d[n]$ can be obtained by inverse discrete-time Fourier transform
$$h_d[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} H_d(e^{j\omega})e^{j\omega n}d\omega.$$
The ideal impulse response may be noncausal and infinitely long. The window method obtains a M-order causal FIR approximation $h[n]$ of the ideal system $h_d[n]$ by truncating (and smoothing) the ideal impulse response by a given window $w[n]$.

$$h[n] = h_d[n]w[n] \quad w[n] = \begin{cases} \text{defined}, & 0 \le n \le M, \\ 0 & \text{otherwise.} \end{cases}$$

Ideal Linear Phase Impulse Response for Window Method

Lowpass Linear Phase Impulse Response for Window Method

Suppose a lowpass FIR filter is specified with cutoff frequency ω_c ($0 < \omega_c < \pi$), symmetric, and order M (even or odd). Then the desired frequency response is

$$H_d(e^{j\omega}) = \begin{cases} 1\, e^{-j\omega M/2} & |\omega| \le \omega_c, \\ 0 & \omega_c < |\omega| \le \pi. \end{cases}$$

By inverse discrete-time Fourier transform, the desired impulse response is
$$h_d[n] = \frac{\sin(\omega_c(n-M/2))}{\pi(n-M/2)}.$$

Highpass Linear Phase Impulse Response for Window Method

Suppose a highpass FIR filter is specified with cutoff frequency ω_c ($0 < \omega_c < \pi$), symmetric, and even order M. Then the desired frequency response is

$$H_d(e^{j\omega}) = \begin{cases} 0 & |\omega| < \omega_c, \\ 1\, e^{-j\omega M/2} & \omega_c \le |\omega| \le \pi. \end{cases}$$

By inverse discrete-time Fourier transform, the desired impulse response is
$$h_d[n] = \frac{\sin(\pi(n-M/2)) - \sin(\omega_c(n-M/2))}{\pi(n-M/2)}.$$

On the other hand, suppose a highpass FIR filter is specified with cutoff frequency ω_c ($0 < \omega_c < \pi$), antisymmetric, and odd order M. Then the desired frequency response is

$$H_d(e^{j\omega}) = \begin{cases} -j\, e^{-j\omega M/2} & -\pi \le \omega \le -\omega_c, \\ 0 & |\omega| < \omega_c, \\ j\, e^{-j\omega M/2} & \omega_c \le \omega \le \pi. \end{cases}$$

By inverse discrete-time Fourier transform, the desired impulse response is
$$h_d[n] = \frac{\cos(\pi(n-M/2)) - \cos(\omega_c(n-M/2))}{\pi(n-M/2)}.$$

Bandpass Linear Phase Impulse Response for Window Method

Suppose a bandpass FIR filter is specified with lower cutoff frequency ω_l, upper cutoff frequency ω_u, ($0 < \omega_l < \omega_u < \pi$), symmetric, and order M (even or odd). Then the desired frequency response is

$$H_d(e^{j\omega}) = \begin{cases} 1\, e^{-j\omega M/2} & \omega_l \le |\omega| \le \omega_u, \\ 0 & |\omega| < \omega_l \text{ and } \omega_u < |\omega| \le \pi. \end{cases}$$

By inverse discrete-time Fourier transform, the desired impulse response is
$$h_d[n] = \frac{\sin(\omega_u(n-M/2)) - \sin(\omega_l(n-M/2))}{\pi(n-M/2)}.$$

Suppose a bandpass FIR filter is specified with lower cutoff frequency ω_l, upper cutoff frequency ω_u, ($0 < \omega_l < \omega_u < \pi$), antisymmetric, and order M (even or odd). Then the desired frequency response is

$$H_d(e^{j\omega}) = \begin{cases} -j\, e^{-j\omega M/2} & -\omega_u \le \omega \le -\omega_l. \\ 0 & |\omega| < \omega_l \text{ and } \omega_u < |\omega| \le \pi, \\ j\, e^{-j\omega M/2} & \omega_l \le \omega \le \omega_u. \end{cases}$$

By inverse discrete-time Fourier transform, the desired impulse response is
$$h_d[n] = \frac{\cos(\omega_u(n-M/2)) - \cos(\omega_l(n-M/2))}{\pi(n-M/2)}.$$

Bandstop Linear Phase Impulse Response for Window Method

Suppose a bandstop FIR filter is specified with lower cutoff frequency ω_l, upper cutoff frequency ω_u, ($0 < \omega_l < \omega_u < \pi$), symmetric, and even order M. Then the desired frequency response is

$$H_d(e^{j\omega}) = \begin{cases} 0 & \omega_l < |\omega| < \omega_u \ , \\ 1\ e^{-j\omega M/2} & |\omega| \le \omega_l \text{ and } \omega_u \le |\omega| \le \pi \ . \end{cases}$$

By inverse discrete-time Fourier transform, the desired impulse response is

$$h_d[n] = \frac{\sin(\omega_l(n - M/2)) + \sin(\pi(n - M/2)) - \sin(\omega_c(n - M/2))}{\pi(n - M/2)}.$$

Window Functions

For common Rectangular, Bartlett, Hann, Hamming, Blackman, Kaiser windows, please refer to [2].

Rectangular

$$w[n] = \begin{cases} 1, & 0 \le n \le M \\ 0, & \text{otherwise} \end{cases}$$

Bartlett (Triangular)

$$w[n] = \begin{cases} 2n/M, & 0 \le n \le M/2 \\ 2 - 2n/M, & M/2 < n \le M \\ 0, & \text{otherwise} \end{cases}$$

Hann

$$w[n] = \begin{cases} 0.5 - 0.5\cos(2\pi n/M), & 0 \le n \le M \\ 0, & \text{otherwise} \end{cases}$$

Hamming

$$w[n] = \begin{cases} 0.54 - 0.46\cos(2\pi n/M), & 0 \le n \le M \\ 0, & \text{otherwise} \end{cases}$$

Blackman

$$w[n] = \begin{cases} 0.42 - 0.5\cos(2\pi n/M) + 0.08\cos(4\pi n/M), & 0 \le n \le M \\ 0, & \text{otherwise} \end{cases}$$

Blackman Harris

$$w[n] = \begin{cases} 0.355768 - 0.487396\cos(2\pi n/M) + 0.144232\cos(4\pi n/M) - 0.012604\cos(6\pi n/M), & 0 \le n \le M \\ 0, & \text{otherwise} \end{cases}$$

Flat Top

$$w[n] = \begin{cases} \frac{1.0 - 1.93\cos(2\pi n/M) + 1.29\cos(4\pi n/M) - 0.388\cos(6\pi n/M) + 0.0322\cos(8\pi n/M)}{4.6402}, \\ \quad 0 \le n \le M, \\ 0, \quad \text{otherwise} \end{cases}$$

Generalized Cosine

The generalized cosine windows are combinations of sinusoidal sequences with frequencies 0, $2\pi/M$, $4\pi/M$, and so on. Hann, Hamming, Blackman, and Flat Top windows are special cases of the generalized cosine windows.

Given parameters A, B, C, D, E, ..., a generalized cosine window is formulated as

$$w[n] = \begin{cases} \frac{A - B\cos(2\pi n/M) + C\cos(4\pi n/M) - D\cos(6\pi n/M) + E\cos(8\pi n/M) - \cdots}{A + B + C + D + E + \cdots}, & 0 \le n \le M \\ 0, & \text{otherwise} \end{cases}$$

Ready

$$a = \frac{\gamma^2 - 1}{\gamma^2} \quad b = 1 + 0.5 * a^2$$

$$w[n] = \begin{cases} \frac{\frac{2n}{M}(1 + 0.5a^2 cos(\frac{4\pi(n - \lfloor M/2 \rfloor)}{M})) + \frac{a}{\pi}(1 - \frac{a}{4})sin(\frac{4\pi|n - \lfloor M/2 \rfloor|}{M})}{b}, & 0 \le n \le \lfloor M/2 \rfloor \\ \frac{(2 - \frac{2n}{M})(1 + 0.5a^2 cos(\frac{4\pi(n - \lfloor M/2 \rfloor)}{M})) + \frac{a}{\pi}(1 - \frac{a}{4})sin(\frac{4\pi|n - \lfloor M/2 \rfloor|}{M})}{b}, & \lfloor M/2 \rfloor + 1 \le n \le M \end{cases}$$

The parameter γ is specified by user and should be $0 < \gamma \le 5$.

Kaiser

The kaiser window is defined as

$$w[n] = \begin{cases} \frac{I_0(\beta\{1 - [(n - M/2)/(M/2)]^2\}^{1/2})}{I_0(\beta)}, & 0 \le n \le M \\ 0, & \text{otherwise.} \end{cases}$$

$I_0(\cdot)$ represents the zeroth-order modified Bessel function of the first kind

$$I_0(x) = 1 + \sum_{k=1}^{\infty} \left(\frac{(x/2)^k}{k!}\right)^2.$$

Based on [2], let δ denote the passband and stopband ripple (if passband and stopband ripples are different, choose the smaller one); let w_p denote the passband cutoff frequency (in radius), i.e., the highest frequency such that $|H(e^{j\omega})| \ge 1 - \delta$; let w_s denote the stopband cutoff frequency (in radius), i.e., the lowest frequency such that $|H(e^{j\omega})| \le \delta$; and let $\Delta\omega = w_s - w_p$ denote the transition width. Defining $A = -20\log_{10}\delta$, Kaiser determined empirically that the value of β is given by

$$\beta = \begin{cases} 0.1102(A - 8.7), & A > 50, \\ 0.5842(A - 21)^{0.4} + 0.07886(A - 21), & 21 \le A \le 50, \\ 0.0, & A < 21. \end{cases}$$

Furthermore, Kaiser found that to achieve prescribed values of A and $\Delta\omega$, M must satisfy

$$M \ge \frac{A - 8}{2.285\Delta\omega}.$$

IIR Filter Design

SystemVue uses digital (discrete-time) IIR filters to implement analog (continuous-time) filters.
In general, SystemVue IIR filters are designed in the following steps:

1. A specific combination of frequency response {lowpass, highpass, bandpass, bandstop} and design method {Bessel, Butterworth, Chebyshev I, Chebyshev II, Elliptic, Synchronously Tuned} is chosen.
2. The parameters of the chosen filter are specified based on users' requirements.
3. The filter specification is translated into lowpass prototype filter specification.
4. A lowpass prototype analog filter is designed. See *Lowpass Analog Filters* (users) for {Bessel, Butterworth, Chebyshev I, Chebyshev II, Elliptic, Synchronously Tuned} analog filters.
5. The lowpass prototype analog filter is transformed into another lowpass, highpass, bandpass, or bandstop analog filter based on one of the *Analog Frequency Transformation* (users) techniques to meet the specified edge frequency (or bandwidth) requirements.
6. The analog filter is converted into a digital IIR filter based on one of the *Analog to Digital Transformation* (users) techniques.

- *Lowpass Analog Filters* (users)
- *Analog Frequency Transformation* (users)
- *Analog to Digital Transformation* (users)
- *S-Domain Design* (users)

Reference

1. A. Antoniou, *Digital Filters: Analysis and Design*. McGraw Hill, 1979.
2. L. B. Jackson, *Digital Filters and Signal Processing*, 3rd ed. Kluwer Academic Publishers, 1995.
3. L. B. Jackson, "A correction to impulse invariance", *Signal Processing Letters*, IEEE, vol. 7, no. 10, pp. 273-275, Oct. 2000.
4. A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*, 2nd ed. Prentice Hall, 1999.
5. J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms and Applications*, 3rd ed. Prentice Hall, 1995.
6. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*. Prentice Hall, 1975.

Analog Frequency Transformation

A lowpass filter can be transformed into another lowpass, highpass, bandpass, or bandstop filters based on the following analog frequency transformation techniques.

Lowpass to Lowpass

Suppose we have a lowpass prototype filter $H_l(s)$ with passband frequency Ω_p, and we wish to transform it to another lowpass filter $H_{l'}(s)$ with passband frequency Ω'_p. This transformation can be accomplished by

$$s \longrightarrow \frac{\Omega_p}{\Omega'_p}s.$$

The resulting lowpass filter has transfer function $H_{l'}(s) = H_l\Big(\frac{\Omega_p}{\Omega'_p}s\Big)$.

Lowpass to Highpass

Suppose we have a lowpass prototype filter $H_l(s)$ with passband frequency Ω_p, and we wish to transform it to a highpass filter $H_h(s)$ with passband frequency Ω'_p. This transformation can be accomplished by

$$s \longrightarrow \frac{\Omega_p\Omega'_p}{s}.$$

The resulting highpass filter has transfer function

$$H_h(s) = H_l\Big(\frac{\Omega_p\Omega'_p}{s}\Big).$$

Lowpass to Bandpass

Suppose we have a lowpass prototype filter $H_l(s)$ with passband frequency Ω_p, and we wish to transform it to a bandpass filter $H_{bp}(s)$ with lower passband edge frequency Ω_l and upper passband edge frequency Ω_u. This transformation can be accomplished by

$$s \longrightarrow \Omega_p\frac{s^2 + \Omega_u\Omega_l}{s(\Omega_u - \Omega_l)}.$$

The resulting bandpass filter has transfer function

$$H_{bp}(s) = H_l\Big(\Omega_p\frac{s^2 + \Omega_u\Omega_l}{s(\Omega_u - \Omega_l)}\Big).$$

> Note that the order of the bandpass filter will be doubled after this lowpass to bandpass frequency transformation. In bandpass filter design specification, the "order" refers to the order of the prototype lowpass filter.

Lowpass to Bandstop

Suppose we have a lowpass prototype filter $H_l(s)$ with passband frequency Ω_p, and we wish to transform
it to a bandstop filter $H_{bs}(s)$ with lower passband edge frequency Ω_l and upper passband edge frequency Ω_u. This transformation can be accomplished by

$$s \longrightarrow \Omega_p\frac{s(\Omega_u - \Omega_l)}{s^2 + \Omega_u\Omega_l}.$$

The resulting bandstop filter has transfer function

$$H_{bp}(s) = H_l\Big(\Omega_p\frac{s(\Omega_u - \Omega_l)}{s^2 + \Omega_u\Omega_l}\Big).$$

> Note that the order of the bandstop filter will be doubled after this lowpass to bandstop frequency transformation. In bandstop filter design specification, the "order" refers to the order of the prototype lowpass filter.

Analog to Digital Transformation

Analog (continuous-time) filters can be converted into digital (discrete-time) IIR filters by the following techniques.

Impulse Invariance

In impulse invariance, a discrete-time system is defined by sampling the impulse response $h_c(t)$ of a continuous-time system. Under certain conditions (discussed below), the resulting impulse response $h[n]$ of the discrete-time system is "invariant" with respect to the sampled version of the impulse response of the continuous-time system.

Suppose a continuous-time system $H_c(s)$ is causal and stable. Based on the discussion found in [1] and [3], if $H_c(s)$ is band limited and the sampling rate ($1/T$) is high enough (such that the aliasing effect is minimal)

$$H_c(j\Omega) \approx 0 \text{ for } |\Omega| \geq \pi/T,$$

we can approximate the continuous-time system in discrete-time domain

$$H(e^{j\omega}) \approx H_c(j\frac{\omega}{T}) \text{ for } |\omega| \leq \pi$$

by setting the impulse response $h[n]$ of the discrete-time system to

$$h[n] = Th_c(nT) - \frac{T}{2}h_c(0^+)\delta[n].$$

Let

$$H_c(s) = A\frac{\prod_{k=1}^{M}(s - z_k)}{\prod_{k=1}^{N}(s - p_k)}$$

denote the transfer function of an analog filter, where $z_1, z_2, \ldots, z_M$ are M zeros and $p_1, p_2, \ldots, p_N$ are N poles.
Suppose $H_c(s)$ is causal, stable, bandlimited, and $N \geq M+1$, and $p_1, p_2, \ldots, p_N$ are single-order poles, the digital IIR filter $H(z)$ can be obtained by impulse invariance as follows:

$$\begin{aligned}
& H_c(s) \\
\xrightarrow{\text{partial fraction expansion}} \quad & H_c(s) = \sum_{k=1}^{N}\frac{A_k}{s - p_k} \\
\xrightarrow{\text{inverse Laplace transform}} \quad & h_c(t) = \sum_{k=1}^{N}A_k e^{p_k t}u(t) \\
\xrightarrow{\text{sampling}} \quad & h[n] = Th_c(nT) - \frac{T}{2}h_c(0^+)\delta[n] = T\sum_{k=1}^{N}A_k e^{p_k nT}u[n] - \frac{T}{2}(\sum_{k=1}^{N}A_k)\delta[n] \\
\xrightarrow{\text{Z transform}} \quad & H(z) = \frac{T}{2}\sum_{k=1}^{N}\frac{A_k(1 + e^{p_k T}z^{-1})}{1 - e^{p_k T}z^{-1}}
\end{aligned}$$

Due to the band limited restriction and the fact that the impulse invariance technique is only practical for $N \geq M+1$, currently only certain lowpass IIR filter blocks in SystemVue provides impulse invariance option.

Bilinear Transformation

Bilinear transformation maps the entire $j\Omega$-axis ($-\infty \leq \Omega \leq \infty$) in the S-plain to one revolution of the unit circle ($-\pi \leq \omega \leq \pi$) in the Z-plain. Bilinear transformation avoids the aliasing problem, but the transformation from S-domain frequency to Z-domain frequency is **nonlinear**.

Bilinear transformation converts S-domain (continuous-time) transfer function $H_c(s)$ into Z-domain (discrete-time) transfer function $H(z)$ by replacing s in $H_c(s)$ with

$$s = \frac{2}{T}\Big(\frac{1 - z^{-1}}{1 + z^{-1}}\Big),$$

where T is the sampling period of the discrete-time system. The resulting Z-domain transfer function is therefore

$$H(z) = H_c\Big[\frac{2}{T}\Big(\frac{1 - z^{-1}}{1 + z^{-1}}\Big)\Big].$$

The mapping between S-domain frequency Ω and Z-domain frequency ω can be expressed in the following relations:

$$\Omega = \frac{2}{T}\tan(\omega/2),$$

$$\omega = 2\arctan(\Omega T/2).$$

Due to the nonlinearity of bilinear transformation, SystemVue IIR filter blocks **prewarp** the critical frequencies, such as passband frequency and stopband frequency, based on the above equation before designing analog filters. With prewarping, the resulting digital filters will meet the desired specification at the critical frequencies.

Bilinear transformation is used as default in SystemVue to convert analog filters to digital filters.

Lowpass Analog Filters

Bessel

Bessel filters are all-pole filters that are characterized by the S-domain transfer function

$$H(s) = \frac{B_0}{B_N(s)},$$

where $B_N(s)$ is the Nth-order Bessel polynomial, and B_0 is the 0th-order coefficient of $B_N(s)$.

The bessel polynomials can be derived recursively from the relation

$$B_N(s) = (2N-1)B_{N-1}(s) + s^2 B_{N-2}(s)$$

with $B_0(s) = 1$ and $B_1(s) = s + 1$ as initial conditions.

Butterworth

Lowpass Butterworth filters are all-pole filters characterized by the magnitude-squared frequency response $|H(\Omega)|^2 = \frac{1}{1 + (\Omega/\Omega_c)^{2N}}$,
where N is the order of the filter, and Ω_c is the -3dB cutoff frequency in radian.

The poles of the lowpass Butterworth filter are $p_k = \Omega_c e^{j\pi/2} e^{j(2k+1)\pi/2N}$, where $k = 0, 1, ..., N-1$.

Chebyshev I

Chebyshev type I filters are all-pole filters that have equiripple behavior in the passband and monotonic behavior in the stopband. The magnitude-squared frequency response of a Chebyshev type I filter is

$$|H(\Omega)|^2 = \frac{1}{1 + \epsilon^2 T_N^2(\Omega/\Omega_p)},$$

where N is the filter order, Ω_p is the passband frequency, ϵ is a parameter related to the passband ripple r_p by

$$r_p = \sqrt{\frac{1}{1 + \epsilon^2}},$$

and $T_N(x)$ is the Nth-order Chebyshev polynomial defined as

$$T_N(x) = \begin{cases} \cos(N\cos^{-1}x), & |x| \le 1 \\ \cosh(N\cosh^{-1}x), & |x| > 1 \end{cases}$$

The poles of a Chebyshev type I filter lie on an ellipse in the S-plain with major axis

$$r1 = \Omega_p \frac{\beta^2 + 1}{2\beta}$$

and minor axis

$$r2 = \Omega_p \frac{\beta^2 - 1}{2\beta},$$

where

$$\beta = \Big(\frac{\sqrt{1+\epsilon^2} + 1}{\epsilon}\Big)^{1/N}.$$

The poles are located in the S-plain at points

$$p_k = r_2 \cos\phi_k + j\, r_1 \sin\phi_k,$$

where

$$\phi_k = \frac{\pi}{2} + \frac{(2k+1)\pi}{2N} \quad k = 0, 1, ..., N-1.$$

Chebyshev II

Chebyshev type II filters have both poles and zeros and exhibit monotonic behavior in the passband and equiripple behavior in the stopband. The magnitude-squared frequency response of a Chebyshev type II filter is

$$|H(\Omega)|^2 = \frac{1}{1 + \epsilon^2 [T_N^2(\Omega_s/\Omega_p)/T_N^2(\Omega_s/\Omega)]},$$

where N is the filter order, Ω_p is the passband frequency, Ω_s
is the stopband frequency, ϵ is a parameter related to the passband attenuation a_p as

$$a_p = \sqrt{\frac{1}{1 + \epsilon^2}},$$

and $T_N(x)$ is the Nth-order Chebyshev polynomial as described above.

The poles of a Chebyshev type II filter are located in the S-plain at points

$$p_k = \frac{\Omega_s x_k}{x_k^2 + y_k^2} + j\frac{\Omega_s y_k}{x_k^2 + y_k^2},$$

where

$$x_k = \frac{\beta^2 - 1}{2\beta}\cos\phi_k,$$

$$y_k = \frac{\beta^2 + 1}{2\beta}\sin\phi_k,$$

and the parameter β is related to the stopband ripple δ by

$$\beta = \Big(\frac{1 + \sqrt{1 - \delta^2}}{\delta}\Big)^{1/N}.$$

The zeros of a Chebyshev type II filter are located on the imaginary axis at points

$$z_k = j\frac{\Omega_s}{\sin\phi_k}.$$

In the above equations,

$$\phi_k = \frac{\pi}{2} + \frac{(2k+1)\pi}{2N} \quad k = 0, 1, ..., N-1.$$

Elliptic

Elliptic filters have equiripple behavior in both the passband and stopband. This class of filters have both poles and zeros and is characterized by the magnitude-squared frequency response

$$|H(\Omega)|^2 = \frac{1}{1 + \epsilon^2 U_N^2(\Omega/\Omega_p)},$$

where $U_N(x)$ is the Jacobian elliptic function of order N, and ϵ is a parameter related to the ripple.

Interested users may find more information on this topic in *Reference 1* (users), where the author provides detailed derivations.

Synchronously Tuned

Synchronously Tuned filters are all-pole filters with all the poles are located at the same point on the negative real axis in the S-plain. A Synchronously Tuned filter is characterized by the S-domain transfer function

$$H(s) = \frac{|p|^N}{(s-p)^N},$$

where p is the pole, and N is the order of the filter.

Given passband frequency Ω_p and passband attenuation a_p, the pole of the Synchronously

Tuned filter can be derived as

$$p = -\frac{\Omega_p}{\sqrt{10^{a_p/(10N)} - 1}}.$$

S-Domain Design

S-Domain design is a different IIR filter design approach. In S-Domain design, users specify the S-Domain poles $p_1, p_2, \ldots, p_M$ and zeros $z_1, z_2, \ldots, z_M$ of the system. To ensure the resulting IIR transfer function is causal, stable, and real coefficients,

- all poles must lie in the left-half of the s-plane,
- the number of zeros must be less than or equal to the number of poles, and
- complex poles must occur in complex conjugate pairs, and complex zeros must occur in complex conjugate pairs.

The S-domain pole-zero system is then transformed into Z-domain transfer function by either bilinear transformation or impulse invariance. Bilinear transformation will result in a Z-domain transfer function that is **not a linear** mapping of the S-domain pole-zero system, see *Bilinear Transformation* (users). On the other hand, impulse invariance restricts the S-domain pole-zero system to be bandlimited, see *Impulse Invariance* (users), and may suffer from implementation difficulty in multiple-order poles.

Equations

Equations are a powerful tool that enable post processing of data, control over inputs to simulations, and definition of user-defined custom models.

Contents

Automatic Calculation

If an Equation object is set to Auto-Calculate, the equations are always kept up to date whenever a value is requested from them. This is desirable when the equation block defines variables that you use in part parameters on a schematic: when you change these values, you want the part parameters that use them to update. **However, sometimes this is undesirable.** If, for example, you are using an Equation block to import data from a file or to transfer data to and from an instrument, you do not want the Equations to calculate unless you specifically tell them to. In these cases, you should disable Auto-Calculate. The Auto-Calculate toolbar button located on the *Equation Toolbar* (users) toggles automatic calculation on and off.

There are some cases where you probably want to DISABLE automatic recalculation of an equation block: equations which do file I/O or TCP/IP communications, equations which run simulations via the *runanalysis* function, equations that do time-consuming processing.

If you disable *Automatic Calculation* , the only way to recalculate the equation is manually with the calculate button, or with the F5 or Ctrl+G keyboard shortcuts. Equations that have *Automatic Calculation* turned off will not update during simulations. As mentioned before, you would normally disable *Automatic Calculation* for Math Language equations that control hardware, for example, so the hardware doesn't get re-controlled every time a variable changes.

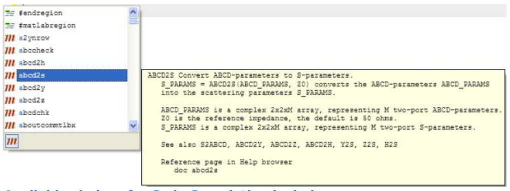
If you disable Auto-Calculate in an equation that is a function definition, the function won't exist until you manually calculate the equation.

Code Completion

Code Completion provides quick access to valid keywords, variables, functions or structure members via the Code Completion Window (as shown below).

To trigger Code Completion widow

Automatically trigger Code Completion
Code Completion window will appear automatically listing all available choices and their brief descriptions while you are typing in the Script Editor.



Available choices for Code Completion include

1. Keywords
2. Built-in variables (available outside Matlab region)
3. User defined variables (available outside Matlab region)
4. Built-in functions (available outside Matlab region)
5. User defined functions (available outside Matlab region)
6. Structure members (available outside Matlab region)
7. Matlab functions (available inside Matlab region)

When a "." is typed after the name of an structure variable, a Code Completion window appears automatically listing all members for that variable. For example "structureVar" is a structure variable contains two members named "a" and "b", when typed "." after "structureVar", its members will be listed in the Code Completion window.

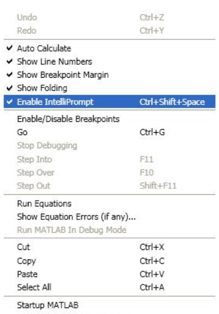
Manually trigger Code Completion
With the edit cursor in the Script Editor field, pressing "Ctrl + SPACE" will trigger the Code Completion window. All available choices will be listed in the Code Completion window. Please be noticed that Matlab functions do not support manually trigger for performance consideration.

To complete code

With the Code Completion window displaying, double clicking a choice, or hitting the Enter key with that choice selected will choose the choice to complete the expression. You can also type in the choice manually. In this case, the closest matching choice will be highlighted while you are typing. In the following figure, when *con* is typed, *conj* is highlighted.

Note

Code Completion is enabled by default. You can disable it by selecting "Enable IntelliPrompt" in the context menu of Script Editor.



Debugging Equations

A fully featured and intuitive debugger is built-in to the equation editing user interface.

Using Breakpoints and Single-Stepping

You can use the *Equation Toolbar* (users) or its associated keyboard shortcuts in the equation script editor to set breakpoints and to step through your code one line at a time. Breakpoints can be set both in equations contained in the workspace and in a model's equations (eg. sub-circuits). In all cases, evaluation of equations will be halted when a breakpoint is hit. The user may then execute statements line by line using single-stepping, abort execution, or continue execution until the next breakpoint is hit.

- **Workspace Equations**: to run the equations click the Go button in the *Equation Toolbar* (users). If a workspace equation is set to Auto-Calculate, they will calculate whenever something they are dependent on triggers a calculation. If any breakpoints are set, the evaluation of the equations is halted and the user interface is brought to the front, clearly marking what line of code the equation processor is currently halted at.

It is important to keep in mind that an equation block may be calculated may times due to various factors, such as a simulator changing a variable. The evaluation of the equations will halt whenever the breakpoint is hit. Typical scenarios include:

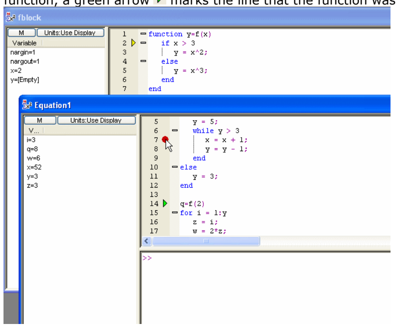
- **Equations in sub-circuit models**: the breakpoint will be hit once per run of the simulator *except* when the equation is dependent on the simulator independent variable.

- **Equations in a Math Language block**: the breakpoint will be hit at each 'tic' of the simulator as data is delivered to the block.

Setting Breakpoints

Click the Breakpoint Margin at the line you wish to set a breakpoint at in the script editor window to toggle a breakpoint on/off. A red dot will appear when the breakpoint is on. The Breakpoint margin is located between the Line Number and Folding margins. You may also set a breakpoint at the current line by using the Ctrl+B keyboard shortcut or clicking the Add/Remove Breakpoint toolbar button.

When the equation processor hits a breakpoint, the current line it is halted at will display a yellow arrow in the breakpoint margin as can be seen in the picture below. At this point you may single-step, step-into functions, continue, or abort execution. If you step-into a function, a green arrow marks the line that the function was called from.

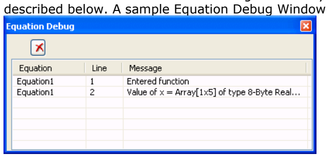


Using Debug Print functions

The debug print functions shown below produce lines of debug text in the Equation Debug docking window.
Please note that debug lines will only appear in the window after the simulator runs, due to current multi-threading locks.

Equation Debug docking window

The Equation Debug docking window can be shown/hidden using the Edit/View/Docking Windows/Equation Debug menu path or using the show/hide dockers button on the main toolbar. This window has a list of debug lines that your equations generate using functions described below. A sample Equation Debug Window is shown here.



Debug Functions

There are two functions available (in both Engineering Language and Mathematics Language) for writing to the Equation Debug docking window. Both functions add lines to the Equation Debug Docking Window so you can trace progress as the program runs. The code samples are written in Mathematics Language.

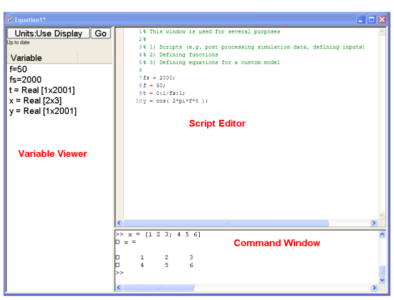
1.
```
dbg_print( 'Message' )
dbg_print( 'Message', 'Equation')
dbg_print( 'Message', 'Equation', Line)
```
   prints the Message in the Equation Debug window. The Equation and Line parameters may be omitted, in which case the equation engine will attempt to auto-detect which equation set and line number called the function.

```
dbg_showvar( 'Message', Variable)
```

prints Message=VariableValue in formatted output

Equations User Interface

The following image shows a typical Equations window:



The Equations window has three subwindows:

- The **Variable Viewer**, located on the left.
- The **Script Editor**, located on the upper right.
- The **Command Window**, located on the lower right.

The Equations window also has an associated toolbar, see *Equation Toolbar* (users).

> Among the simplest application of equations is to define a variable in the Script Editor area (upper right), such as myvar=123 (then press "Go")
> Then myvar can be entered into component properties on the schematic, to drive component values.
> Entering myvar=?123 (ie. adding the question mark) makes the value myvar tunable in the tune window.
> After pressing "Go", the variable value should appear in the Variable Viewer (left side). If nothing appears in the Variable Viewer after pressing "Go", this usually indicates some problem with equation syntax.
> Typically the error messages window will provide some clues.

Variable Viewer

The Variable Viewer displays any variables that currently exist in the Equations object. If the variable is a scalar, the value is displayed. If the variable is an array, the type and dimensions of the array are displayed.

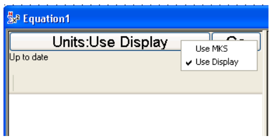
If you right-click on any variable displayed in the Variable Viewer, you will be presented with a menu containing options to plot the variable on a graph or display it in a table. If you wish to see the variable's value without creating a table, you can do so in the

Command Window, as discussed below.

The following buttons are located at the top of the Variable Viewer window: Units and Go.

The Units button allows you to define how the values of the variables defined in the Equations page are to be interpreted when used elsewhere, such as part parameters. If Units is set to "Use MKS", then the values of the variables will be treated as if they were in MKS units. Any unit specified when the variable is used (e.g. in a part parameter) will only be used for schematic display purposes and will not further scale the value. If, on the other hand, Units is set to "Use Display", then the units will be defined where the value is actually used. In this case any unit specified when the variable is used (e.g. in a part parameter) will be used to scale the variable value as well as for schematic display purposes.

For example, assume a variable F=10000 is defined in an Equations page that has Units set to Use MKS. If this variable is used to set the F3dB parameter of a part and the unit of the F3dB parameter is set to KHz, then the F3dB parameter is set to 10000 (unit is ignored in setting the actual value of the parameter and is only used for schematic display purposes) and it will show (on the schematic) as 10 KHz. If the same variable is defined in an Equations page that has Units set to Use Display and used to set the F3dB parameter as described above, then the F3dB parameter will be set to 10000000 (the last three zeros is because of the scaling provided by the KHz unit) and it will show (on the schematic) as 10000 KHz.



- Set Units to Use Display only when you want a unit to be attached wherever the variable is used.
- Set Units to Use MKS in the Equation pages of models to ensure their portability regardless of an end user's unit preferences. If the Units for the Equations page of a model are not set to Use MKS, the model will not function properly when the units of its parameters are changed from their default setting.

The Go button provides an easy way to force execution of the equations. Its function is equivalent to the Go button on the *Equation Toolbar* (users).

If you want the variable block to be cleared each time the equations are executed, the first line of your equations should be the *clear* statement.

Script Editor

The Script Editor is used to type in a sets of equation statements to be executed. More specifically, the Script Editor window is used to:

- Post-Process data, or define variables as inputs to be used elsewhere.
- Create user-defined functions.
- Define equations inside a Model.

The Script Editor includes Find and Replace support, accessible from the Edit menu or with the Ctrl+F or Ctrl+H keys, respectively.

New If you want context sensitive help on a function, select the keyword and press F1 in the Script Editor.
New Use Ctrl_MouseWheel to zoom in and out on the equations Script Editor.

Command Window

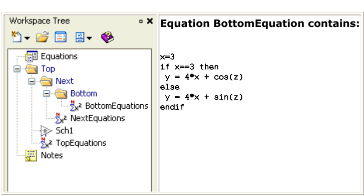
The Command Window is used to execute statements line-by-line. It interacts with the same variables that are visible to the Script Editor. It is a useful debugging tool since the contents of a variable can be displayed here.

If an assignment statement does not end in a semicolon, the results of that assignment are outputted in the Command Window, as can be seen in the above figure. If an assignment statement does end in a semicolon, then the dump of the contents of the result is suppressed.

Any errors or warnings caused by executing a line in the Command Window are outputted to the Command Window.

Hierarchy in Equations

Equations obey hierarchy as defined by their place in the Workspace Tree. Note that this is true for Equation objects as well as equations that are embedded inside a Design object (ie. an Equation tab in a Design).



In the example above, the value of *z* will be coming from another equation set (NextEquations or TopEquations) to execute without errors. The Equations engines look up the workspace hierarchy until the value of *z* is found, otherwise an error is reported.

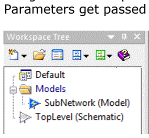
In other words, in this example, if *z* is defined in NextEquations, then *z* will come from there. If NextEquations does not define *z* then the Equations engine looks up another folder level to TopEquations for *z* .\

Equations on the same level of hierarchy should all be visible to each other.

Design-time vs. Run-time hierarchy

The above discussion of hierarchy is called design-time hierarchy, because while you are not simulating, the workspace tree defines the scoping of variables. However, when you run a simulation, the situation can be different.

Suppose, for example, that we have 2 designs as shown in the below picture. Both designs have an Equation tab (and possibly a Parameters tab, which is equivalent, since Parameters get passed into the design's Equations at run-time).



In the situation shown in the above picture, when you are NOT running a simulation (ie. you are in design-time), the design called SubNetwork will be able to see variables that are defined in the Equations tab of the design called TopLevel, simply because SubNetwork is located in a folder beneath the level of hierarchy that TopLevel is in. However, suppose that, as shown, SubNetwork defines a subnetwork model. Also suppose that the schematic in TopLevel defines an instance of SubNetwork (ie. it has a part that references the SubNetwork model). When you run a simulation (ie. during run-time), a Model hierarchy is defined in which SubNetwork is a child of TopLevel, since an instance of a SubNetwork model is instantiated inside of TopLevel. Because of this, SubNetwork can see all of TopLevel's variables. This is what makes the passing of parameters from TopLevel to SubNetwork possible.

It is important to note that when you are editing a design (ie. you are in design-time), the values of parameters you see in your design are those calculated using the design-time hierarchy. For example, if you define a Design that contains Parameters, and you use one of those parameters inside your Design, you will see the value of that parameter correspond to the "Default" value of the Parameter that you defined in the Parameters tab. This is, of course, not necessarily the value that will be seen at run-time when you run a simulation, since that depends on the run-time hierarchy defined by the topology of the network you are simulating.

Using Math Language

Math Language, along with most of its built-in functions, was designed to be compatible with m-file script syntax.

- For a full description see *Using Math Language* (users).
- For a complete function reference see *Math Language Function Reference* (users)

Math Language Function Reference

To go directly to entries that belong to a category, select one of the following: *Basic* (users), *Communications* (users), *Signal Processing* (users)
To go directly to entries that start with a specific letter, select one of the following: A, B, C , D, E, F, G, H, I, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Z.

Function Name	Description
abs (users)	absolute value or magnitude
acos (users)	inverse cosine, in radians
acosd (users)	inverse cosine, in degrees
acosh (users)	inverse hyperbolic cosine
acot (users)	inverse cotangent
acotd (users)	inverse cotangent, in degrees
acoth (users)	inverse hyperbolic cotangent
acsc (users)	inverse cosecant
acscd (users)	inverse cosecant, in degrees
acsch (users)	inverse hyperbolic cosecant
all (users)	true if all parts in a vector are nonzero
angle (users)	phase of a complex number, in radians
any (users)	true if any part in a vector is nonzero
asec (users)	inverse secant, in radians
asecd (users)	inverse secant, in degrees
asech (users)	inverse hyperbolic secant
asin (users)	inverse sine, in radians
asind (users)	inverse sine, in degrees
asinh (users)	inverse hyperbolic sine
atan (users)	inverse tangent, in radians
atan2 (users)	4-quadrant inverse tangent, in radians
atand (users)	inverse tangent, in degrees
atanh (users)	inverse hyperbolic tangent
alignsignals (users)	align two signals by delaying earliest signal
awgn (users)	add white Gaussian noise to signal
bartlett (users)	Bartlett Window
blackman (users)	Blackman Window
butter (users)	Butterworth filter designer
bi2de (users)	convert binary vectors to decimal
bilinear (users)	parameter transformation from analog filter to digital filter
buttord (users)	butterworth filter order and cutoff frequency calculation
ceil (users)	smallest integer greater than or equal to argument
cheby1 (users)	Chebyshev type 1 filter designer
cheby2 (users)	Chebyshev type 2 filter designer
class (users)	data-type (class name) of argument
conj (users)	complex conjugate
conv (users)	linear convolution (or polynomial multiplication)
cos (users)	cosine of a radian-valued argument
cosd (users)	cosine of a degree-valued argument
cosh (users)	hyperbolic cosine
cot (users)	cotangent of a radian-valued argument
cotd (users)	cotangent of a degree-valued argument
coth (users)	hyperbolic cotangent
csc (users)	cosecant of a radian-valued argument
cscd (users)	cosecant of a degree-valued argument
csch (users)	hyperbolic cosecant
convdeintrlv (users)	permute data with specified shift register group
convenc (users)	convolutionally encode binary data
convintrlv (users)	permute data with specified shift register group
crcdec (users)	cyclic redundancy check decoder
crcenc (users)	cyclic redundancy check encoder
cheb1ord (users)	minimum order calculation for Chebyshev Type I filter
cheb2ord (users)	minimum order calculation for Chebyshev Type II filter
dbg_print (users)	output to equation debug window
dbg_showvar (users)	output contents of a variable to equation debug window
deconv (users)	deconvolution (or polynomial division)
dec2hex (users)	decimal to hexadecimal conversion
diag (users)	create diagonal matrix or extract diagonal of a matrix
diff (users)	difference (or approximate derivative)
de2bi (users)	decimal numbers to binary vectors
deintrlv (users)	reorder data back with specified permutation table
depuncture (users)	restores erasures based on puncture pattern
downsample (users)	downsample input signal
dpskdemod (users)	differential phase-shift keying demodulation
dpskmod (users)	differential phase-shift keying modulation
eig (users)	eigenvalues and eigenvectors of a matrix
ellip (users)	elliptic or cauer filter designer
equalize (users)	equalize signal using Equalizer
erf (users)	error function
erfc (users)	complementary error function
error (users)	posts to error log or output error to command window
exist (users)	check the existance of a variable or a builtin function
exp (users)	exponential
eye (users)	construct identity matrix
eyediag (users)	build an eye diagram from time data
false (users)	logical false
fclose (users)	close a file or stream
fft (users)	Discrete Fourier Transform (DFT)
fgets (users)	read a line from a file, keep newline
filter (users)	one dimensional digital filtering
find (users)	indices of nonzero parts
findstr (users)	find a string within another string

firls (users)	multiband least square FIR filter design
firrcos (users)	raised cosine FIR Filter design
fix (users)	round toward zero
floor (users)	largest integer less than or equal to argument
fopen (users)	open file or stream
fread (users)	read binary data from a file or stream
fprintf (users)	write formatted text to a file or stream
fscanf (users)	read formatted text from a file or stream
fwrite (users)	write binary data to a file or stream
finddelay (users)	estimate delay(s) between signals
fftfilt (users)	FFT-based FIR filtering using overlap-add method
gausswin (users)	Gaussian Window
getindep (users)	returns the string property containing the path to the independent value of a variable x. (ie. the reference to the independent variable)
getindepvalue (users)	returns the single independent value of a variable x.
getmatlabvariables (users)	get the value of a variable list from MATLAB
getunits (users)	Returns an integer corresponding to the units of a variable x. This integer may be used by setunits.
getvariable (users)	get the value of a variable from a dataset
gaussfir (users)	Gaussian FIR Pulse-Shaping Filter Design
grpdelay (users)	group delay of IIR filter
hamming (users)	Hamming Window
hann (users)	Hann Window
hex2dec (users)	hexadecimal to decimal conversion
hilbert (users)	compute the analytic signal from a real data vector
histc (users)	histogram count
ifft (users)	Inverse Discrete Fourier Transform (IDFT)
imag (users)	imaginary part of a complex number
impz (users)	Impulse response of IIR digital filter
inf (users)	infinity
interp1 (users)	one dimensional interpolation
ischar (users)	true if argument is of type character array
isempty (users)	true if argument is empty or array with a dimension of length 0
isequal (users)	true if arrays contain equal values, ignoring NaNs
isfinite (users)	true for finite parts
isfloat (users)	true if argument is a floating point scalar or array
isinf (users)	true for infinite parts
isinteger (users)	true if argument is an integer scalar or array
islogical (users)	true if argument is a logical scalar or array
isnan (users)	true for NaN parts
isreal (users)	true if argument is a real-valued scalar or array
isscalar (users)	true if argument is a scalar
isstr (users)	true if argument is a character array
interp (users)	resample input at a higher rate with lowpass filter
kaiser (users)	kaiser window
kaiserord (users)	Parameters that specify a kaiser window
length (users)	length of a vector
linspace (users)	construct linearly spaced vector
log (users)	natural logarithm
log2 (users)	Base-2 logarithm
log10 (users)	Base-10 logarithm
logspace (users)	construct logarithmically spaced vector
lu (users)	LU matrix factorization
lp2bp (users)	transform lowpass filter to bandpass filter
lp2bs (users)	transform lowpass filter to bandstop filter
lp2hp (users)	transform lowpass filter to highpass filter
lp2lp (users)	lowpass filter with normalized frequency to desired frequency
max (users)	largest value of a vector
mean (users)	arithmetic mean of a vector
median (users)	median of a vector
min (users)	smallest value of a vector
mkdir (users)	make directory
mod (users)	modulus after division
mode (users)	mode (most frequent value) of a vector
matdeintrlv (users)	reorder data by filling matrix by columns and emptying it by rows
matintrlv (users)	reorder data by filling matrix by rows and emptying it by columns
muxdeintrlv (users)	restore ordering of data with specified shift register group
muxintrlv (users)	reorder data with specified shift register group
nan (users)	Not-a-Number
num2str (users)	convert number to a character array
numel (users)	total number of parts in an array
noisebwlv (users)	equivalent two-sided noise bandwidth of lowpass filter
oct2dec (users)	convert octal to decimal numbers
poly2trellis (users)	convert convolutional code polynomials to trellis description
puncture (users)	Erase specified symbols based on puncture pattern
phasedelay (users)	return phase delay vector for digital filter
qamdemod (users)	Quadrature amplitude demodulation
qammod (users)	Quadrature amplitude modulation
qfunc (users)	Q function
qfuncinv (users)	inverse Q function
rand (users)	uniformly distributed random numbers between 0 and 1
randn (users)	Normally (Gaussian) distributed random numbers
rcosflt (users)	Filter input signal with (sqrt) raised cosine filter
real (users)	real part of a complex number
rectwin (users)	Rectangular Window
rem (users)	remainder after division
resample (users)	Change sampling rate by rational factor
reshape (users)	change dimensions of an array
roots (users)	roots of a polynomial

round (users)	round towards nearest integer
runanalysis (users)	Run an analysis in the workspace tree. Useful for scripting simulations.
randerr (users)	generate bit error patterns
randint (users)	generate uniformly distributed random integers
randsrc (users)	generate random matrix using prescribed alphabet
rectpulse (users)	rectangular pulse shaping
rsdec (users)	reed-Solomon decoder
rsenc (users)	reed-Solomon encoder
sec (users)	secant of a radian-valued argument
secd (users)	secant of a degree-valued argument
sech (users)	hyperbolic secant
setindep (users)	set the independent reference for a swept dependent variable to indepvar(s). A minimum of two arguments is required. This function can be used to remove all independent values of a variable by passing in a blank string for the second argument.
setmatlabvariables (users)	define MATLAB variables and set SystemVue variables' value to MATLAB
setvariable (users)	write a value to a variable in a dataset
setunits (users)	sets a variable to have units specified by unit. The unit may be an integer or a string. Integer units correspond to the units returned by the getunits function. Units do not change the underlying value of a variable, but rather, just change how the value is displayed. Example: setunits('freqaxis', 'MHz')
sign (users)	signum
sin (users)	sine of a radian-valued argument
sinc (users)	sinc function (sin(pi*x) / (pi*x))
sind (users)	sine of a degree-valued argument
sinh (users)	hyperbolic sine
size (users)	dimensions of an array
skewness (users)	skewness of a vector
sort (users)	sort a vector in ascending or descending order
spline (users)	cubic spline interpolation
sqrt (users)	square root
sscanf (users)	read formatted text from a string
ss2tf (users)	Convert state-space filter parameters to transfer function form
ss2zp (users)	Convert state-space filter parameters to zero-pole-gain form
std (users)	standard deviation of a vector
str2num (users)	convert a string to a number
strcmp (users)	case-sensitive string comparison
strcmpi (users)	case-insensitive string comparison
strncmp (users)	compare first N characters of a string (case-sensitive)
strncmpi (users)	compare first N characters of a string (case-insensitive)
struct (users)	construct a structure array
sum (users)	sum of the parts of a vector
svd (users)	matrix singular value decomposition
symerr (users)	compute number of symbol errors and symbol error rate
sftrans (users)	transform of lowpass filter to other type filter
square (users)	Square wave generation
tan (users)	tangent of a radian-valued argument
tand (users)	tangent of a degree-valued argument
tanh (users)	hyperbolic tangent
tcpip (users)	construct tcpip stream object for TCP/IP communications
tf2ss (users)	Convert transfer function filter parameters to state-space form
tf2zp (users)	convert transfer function filter parameters to zero-pole-gain form
toeplitz (users)	construct Toeplitz matrix
true (users)	logical true
turbodec (users)	compute number of symbol errors and symbol error rate
turboenc (users)	inverse Q function
triang (users)	coefficients of a triangular window
using (users)	sets the current context in an equation block to the dataset called Dataset
upfirdn (users)	Upsample by zero inserting, filtering and downsampling a signal
upsample (users)	Upsample input signal by inserting R-1 zeros between elements
var (users)	variance of a vector
vitdec (users)	convolutionally decodes binary stream using Viterbi algorithm
warning (users)	posts a warning to error log or output warning to command window
wgn (users)	generates white Gaussian noise
xcorr (users)	cross correlation
xor (users)	logical exclusive-OR
zp2ss (users)	Convert zero-pole-gain filter parameters to state-space form
zp2tf (users)	Convert zero-pole-gain filter parameters to transfer function form

abs

Syntax
y = abs(x)

Definition
This function takes the absolute value of a real variable or the magnitude of a complex variable. It operates on an part-by-part basis on arrays.

Examples:

Formula	Result
abs(-1.5)	1.5
abs(complex(1,1))	1.414
abs([-1;-2;3])	[1;2;3]

Compatibility
scalars, vectors, arrays

acos

Syntax
y = acos(x)

Definition
This function returns the inverse cosine of the angular value x, in radians expressed in the MKS range [0, PI]. It operates on an part-by-part basis on arrays. It cannot accept a complex valued variable.

Examples:

Formula	Result	or
acos(0)	1.571	PI/2
acos(1)	0	0
acos(-1)	3.141	PI
acos(.707)	0.786	PI/4
acos([-.707 0 1])	[2.356 1.571 0]	[3*PI/4 PI/2 0]

Compatibility
Real valued scalars, vectors, arrays

See Also
acosd (users)
acosh (users)
cos (users)
cosd (users)
cosh (users)

acosd

Syntax
y = acosd(x)

Definition
This function returns the inverse cosine of the angular value x, in radians expressed in the range [0, 180]. It operates on an part-by-part basis on arrays. It cannot accept a complex valued variable.

Examples:

Formula	Result
acosd(0)	90
acosd(1)	0
acosd(-1)	180
acosd(.707)	45
acosd([-.707 0 1])	[135 90 0]

Compatibility
Real valued scalars, vectors, arrays

See Also
acos (users)
acosh (users)
cos (users)
cosd (users)
cosh (users)

acosh

Syntax
y = acosh(x)

Definition
This function returns the inverse of the hyperbolic cosine of the number x. It operates on an part-by-part basis on arrays. It cannot accept a complex valued variable.

```
cosh(x) = log( x + sqrt(  x^2 - 1))
```

Examples:

Formula	Result
acosh(1)	0
acosh(10)	2.993
acosh(0)	NaN

Compatibility
Real valued scalars, vectors, arrays

See Also

acos (users)
acosd (users)
cos (users)
cosd (users)
cosh (users)

acot

Syntax
y = acot(x)

Definition
This function returns the inverse co-tangent of the angular value x, in radians expressed in the MKS range [0, PI]. It operates on an part-by-part basis on arrays. It cannot accept a complex valued variable.

Formula	Result	or
acot(1.732)	0.5236	PI/6
acot(0.577)	1.0472	PI/3

Compatibility
Real valued scalars, vectors, arrays

See Also:

acotd (users)
acoth (users)
cot (users)
cotd (users)
coth (users)

acotd

Syntax
y = acotd(x)

Definition
This function returns the inverse co-tangent of the angular value x, in radians expressed in the range [0, 180]. It operates on an part-by-part basis on arrays. It cannot accept a complex valued variable.

Formula	Result
acotd(1.732)	30
acotd(0.577)	60

Compatibility
Numeric scalars, vectors, arrays

See Also:
acot (users)
acoth (users)
cot (users)
cotd (users)
coth (users)

acoth

Syntax
y = acoth(x)

Definition
This function returns the inverse of the hyperbolic co-tangent of the number x. It operates on an part-by-part basis on arrays. It cannot accept a complex valued variable.

Compatibility
Real valued scalars, vectors, arrays

See Also:

acot (users)
acotd (users)
cot (users)
cotd (users)
coth (users)

acsc

Syntax
y = acsc(x)

Definition
This function returns the inverse co-secant of the angular value x, in radians expressed in

the MKS range [0, PI]. It operates on an part-by-part basis on arrays. It cannot accept a complex valued variable.

Compatibility
Real valued scalars, Vectors, Arrays

See Also:
acscd (users)
acsch (users)
csc (users)
cscd (users)
csch (users)

acscd

Syntax
y = acscd(x)

Definition
This function returns the inverse co-secant of the angular value x, in degrees expressed in the range [0, 180]. It operates on an part-by-part basis on arrays. It cannot accept a complex valued variable.

Compatibility
Real valued scalars, Vectors, Arrays

See Also:
acsc (users)
acsch (users)
csc (users)
cscd (users)
csch (users)

acsch

Syntax
y = acsch(x)

Definition
This function returns the inverse of the hyperbolic co-secant of the number x. It operates on an part-by-part basis on arrays. It cannot accept a complex valued variable.

Compatibility
Numeric scalars, Vectors, Arrays

See Also:
acsc (users)
acscd (users)
csc (users)
cscd (users)
csch (users)

alignsignals

align two signals by delaying earliest signal

Syntax

[Xa,Ya] = alignsignals(X,Y)

[Xa,Ya] = alignsignals(X,Y,MAXLAG)

[Xa,Ya] = alignsignals(...,'truncate')

[Xa,Ya,D] = alignsignals(...)

Definition

[Xa Ya] = alignsignals(X,Y), X and Y should be vectors and the return Xa and Ya are both column vectors. This function estimates the delay between X and Y via finddelay function and then delay the earlier signal by inserting zeros to align these two signals.

[Xa Ya] = alignsignals(X,Y,MAXLAG), MAXLAG should be a integer scalar ranging from 0 to the larger length of X and Y minus 1. The search range should fall in the range of [-MAXLAG MAXLAG].

[Xa Ya] = alignsignals(...,'truncate'), if X or Y are delayed by inserting some zeros, 'truncate' will cut some tail elements to remain its original length.

Examples

Compatibility

See also
finddelay (users)

all

Syntax
all(data)
all(data, dim)

Definition
This function returns true if all values in a vector are non-zero or logical true, otherwise it returns false. If *data* is a matrix, then this function operates on the columns of data.

The *dim* argument is optional and specifies which dimension to operate along. For example, if *dim* is 1, this function operates on each column of the argument. If the argument is omitted, the first non-singleton dimension is chosen as the dimension to operate along.

Examples:

Formula	Result
all([1 0 0 1 0])	0
all([1 1 1])	1

For **A** = [1 1 0; 1 0 1; 1 1 1];

Formula	Result	Comments
all(A)	[1 0 0]	Returns dim=1 column wise results
all(A, 1)	[1 0 0]	Returns column wise results
all(A, 2)	[0; 0; 1]	Returns row wise results

Compatibility
vectors, arrays

See Also
any (users)

angle

Syntax
y = angle(x)

Definition
This function returns the phase of a complex number, in radians.
This function operates on an part-by-part basis on arrays.

Compatibility
Complex valued scalars, vectors, arrays

Real valued variables are treated as vectors with angular value of zero.

any

Syntax
any(data)
any(data, dim)

Definition
This function returns true if any of the values in a vector are non-zero or logical true, otherwise it returns false. If *data* is a matrix, then this function operates on the columns of data.

The *dim* argument is optional and specifies which dimension to operate along. For example, if *dim* is 1, this function operates on each column of the argument. If the argument is omitted, the first non-singleton dimension is chosen as the dimension to operate along.

Examples:

Formula	Result
all([1 0 0 1 0])	1
all([0 0 0])	0

For **A** = [0 0 1; 0 1 0; 0 0 0];

Formula	Result	Comments
all(A)	[0 1 1]	Returns dim=1 column wise results
all(A, 1)	[0 1 1]	Returns column wise results
all(A, 2)	[1; 1; 0]	Returns row wise results

Compatibility
vectors, arrays

See Also
all (users)

asec

Syntax
y = asec(x)

Definition
This function is the inverse secant, in radians in the range [0, PI].
This function operates on an part-by-part basis on arrays.

Compatibility
Real valued scalars, vectors, arrays

See Also
asecd (users)
asech (users)
sec (users)
secd (users)
sech (users)

asecd

Syntax
y = asecd(x)

Definition
This function is the inverse secant, in degrees.
This function operates on an part-by-part basis on arrays.

Compatibility
Real valued scalars, vectors, arrays

See Also
asec (users)
asech (users)
sec (users)
secd (users)
sech (users)

asech

Syntax
y = asech(x)

Definition
This function returns the inverse hyperbolic secant of the argument.
This function operates on an part-by-part basis on arrays.

Compatibility
Real valued scalars, vectors, arrays

See Also
asecd (users)
sec (users)
secd (users)
sech (users)

asin

Syntax
y = asin(x)

Definition
asin returns the inverse sine of the argument, in radians, between -PI / 2 <= r <= PI / 2.
This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result	or
asin (0)	0	0
asin (1)	1.571	PI/2
asin(-1)	-1.571	-PI/2
asin (.707)	0.786	PI/4
asin (-.707)	-0.786	-PI/4

Compatibility
Real valued scalars, vectors, arrays

See Also
asind (users)
asinh (users)
sin (users)
sind (users)
sinh (users)

asind

Syntax
y = asind(x)

Definition
asind returns the inverse sine of the argument, in degrees, in a range of [-180, 180]. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result	in Radians
asin (0)	0	0
asin (1)	180	PI/2
asin(-1)	-180	-PI/2
asin (.707)	45	PI/4
asin (-.707)	-45	-PI/4

Compatibility
Real valued scalars, vectors, arrays

See Also:
asin (users)
asinh (users)
sin (users)
sind (users)
sinh (users)

asinh

Syntax
y = asinh(x)

Definition
This function returns the inverse hyperbolic sine of the argument, equal to log(x + sqrt(x^2 + 1)). This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
asinh(1)	0.881
asinh(10)	2.998
asinh([0 1 10])	[0 0.881 2.998]

Compatibility
Real valued scalars, vectors, arrays

See Also:
asind (users)
asin (users)
sin (users)
sind (users)
sinh (users)

atan

Syntax
y = atan(x)

Definition
This function returns the inverse tangent of the argument, in radians between -PI/2 < r < PI/2. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
atan(0)	0
atan(1)	0.785
atan([-1 .5 -.5])	[-0.785 0.464 -0.464]

Compatibility
Real valued scalars, vectors, arrays

See Also:
tanh (users)
atan2 (users)
atand (users)
atanh (users)
tan (users)
tand (users)

atan2

Syntax
y = atan2(y, x)

Definition
atan2 returns the 4-quadrant inverse tangent of the argument, in radians. The return value is the same size as the input arrays y and x, and is computed on an part-by-part basis. Either argument may be a scalar, in which case that argument is expanded to be the same size as the other argument. For complex inputs, imaginary parts are ignored.

Examples:

Formula	Result	or
atan2(1, 0)	1.571	pi/2
atan2(1, 1)	0.785	pi/4
atan2([1; 0; -1], -1)	[2.356; 3.142; -2.356]	[3*pi/4; pi; -3*pi/4]

Compatibility
Real valued scalars, vectors, arrays

See Also
atan (users)
tan (users)

atand

Syntax
y = atand(x)

Definition
This function returns the inverse tangent of the argument, in degrees.
This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result	in Radians
atan(0)	0	
atan(1)	45	PI/4
atan(-1)	-45	-PI/4

Compatibility
Real valued scalars, vectors, arrays

See Also:
atan (users)
tan (users)
tand (users)
atanh (users)
tand (users)

atanh

Syntax
y = atanh(x)

Definition
This function returns the inverse hyperbolic tangent of the argument, which is equivalent to 0.5 * log((1 + x) / (1 - x)). This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
atanh(1)	undefined
atanh(.5)	0.549
atanh(-.5)	-0.549
atanh(0)	0

Compatibility
Real valued scalars, vectors, arrays

See Also:
atan (users)
tan (users)
atand (users)
tanh (users)
tand (users)

awgn

add white Gaussian noise to signal

Syntax

Y = AWGN(X,SNR)

Y = AWGN(X,SNR,SIGPWR)

Y = AWGN(X,SNR,'measured')

Y = AWGN(X,SNR,SIGPWR,STATE)

Y = AWGN(X,SNR,'measured',STATE)

Y = AWGN(...,POWERTYPE)

Definition

Y = AWGN(X,SNR) adds white Gaussian noise to signal x. The snr is in dB. If x is complex, awgn adds complex noise. The power of X is 0 dBW by default.

Y = AWGN(X,SNR,SIGPWR) specifies the the X power to SIGPWR dBW

Y = AWGN(X,SNR,'measured') measures the power of input signal before adding noise.

Y = AWGN(X,SNR,SIGPWR,STATE) specifies the state of the random number generator.

Y = AWGN(..., POWERTYPE) is the same as the previous syntaxes with powertype specified. Choices for powertype are 'dBW', 'dBm', and 'linear'.

Examples

Compatibility

See also
wgn (users), *randn* (users)

bartlett

Bartlett window

Syntax

W = bartlett(N)

Definition

This function returns a column vector containing a Bartlett window with N points, N being a positive integer greater than 2. The Bartlett window is characteristically triangular in shape with a base value of 0 and an apex value of 1. When N is odd, the apex is explicitly an part of the window function. When N is even, the apex is not explicitly sampled but rather the two sample points which flank the apex are represented in the returned vector.

Note
bartlett(2), a redundant usage of this function returns [0 0] whereas bartlett(1) returns [1].

Examples:

Formula	Result	Comment
bartlett(13)	[0,1,2,3,4,5,6,5,4,3,2,1,0]/6	(13-1)/2=6 is common divisor
bartlett(14)	[0,1,2,3,4,5,6,6,5,4,3,2,1,0]/6.5	(14-1)/2=6.5 is common divisor

The graph shows how bartlett(14) does not sample the peak value of 1 at 6.5 explicitly but bartlett(13) does.

Compatibility
scalar

See Also
blackman (users), *gausswin* (users), *hamming* (users), *hann* (users), *rectwin* (users)

bi2de

Convert binary vectors to decimal numbers

Syntax

d = bi2de(b)

d = bi2de(b,flg)

d = bi2de(b,p)

d = bi2de(b,p,flg)

Definition

D = bi2de(B) converts binary row vector to positive decimal integer. If B is MB-NB matrix,D should be a MB-1 vector.

D = bi2de(B,FLG), FLG can be 'left-msb' or 'right-msb'. default is 'right-msb'.

D = bi2de(B,P) converts base-P row vector to positive decimal integer.

Examples

Compatibility

See also
de2bi (users)

bilinear

Bilinear parameter transformation from analog filter to digital filter

Syntax

[Zd,Pd,Kd] = bilinear(Za,Pa,Ka,Fs)

[Zd,Pd,Kd] = bilinear(Za,Pa,Ka,Fs,Fp)

[NUMd,DENd] = bilinear(NUMa,DENa,Fs)

[NUMd,DENd] = bilinear(NUMa,DENa,Fs,Fp)

[Ad,Bd,Cd,Dd] = bilinear(Aa,Ba,Ca,Da,Fs)

[Ad,Bd,Cd,Dd] = bilinear(Aa,Ba,Ca,Da,Fs,Fp)

Definition

1. Bilinear transforms analog filter parameters (s-domain) to digital filter equivalent (in z-domain) with equation (without frequency prewarping):

```
H(z) = H(s) |     2   z-1
            | s = - * ---
            |    Ts  z+1
```

or (with frequency prewarping)

```
H(z) = H(s) |        2*pi*Fp      z-1
            | s = ------------- * ---
            |    tan(pi*Fp*Ts)   z+1
```

where Ts is sample period in second (the reciprocal of sample frequency). Prewarping indicates frequency responses match exactly at frequency point Fp (in Hz) before and after mapping.

[Zd,Pd,Kd] = bilinear(Za,Pa,Ka,Fs), where column vectors Za, Pa and scalar Ka are s-domain zeros, poles and gain, column vectors Zd, Pd and scalar Kd are z-domain zeros, poles and gain. H(s) and H(z) are represented as:

```
       Za(s)   [s-Za(1)] * [s-Za(2)] * ... * [s-Za(n)]
H(s) = ----- = ---------------------------------------
      Pa(s)   [s-Pa(1)] * [s-Pa(2)] * ... * [s-Pa(n)]
      Zd(z)   [z-Zd(1)] * [z-Zd(2)] * ... * [z-Zd(n)]
H(z) = ----- = ---------------------------------------
      Pd(z)   [z-Pd(1)] * [z-Pd(2)] * ... * [z-Pd(n)]
```

[NUMd,DENd] = bilinear(NUMa,DENa,Fs), converts s-domain transfer function given by NUMa and DENa to z-domain equivalent. Row vectors NUMa and DENa specify the s-domain coefficients of numerator and denominator, in descending powers of s. Row vectors NUMd and DENd specify the z-domain coefficients of the numerator and denominator, in descending powers of z. H(s) and H(z) are represented as:

```
       NUMa(s)   [NUMa(1) NUMa(2) ... NUMa(n+1)] * s^[n n-1 ... 0].'
H(s) = ------- = ---------------------------------------------------
      DENa(s)   [DENa(1) DENa(2) ... DENa(n+1)] * s^[n n-1 ... 0].'
      NUMd(z)   [NUMd(1) NUMd(2) ... NUMd(n+1)] * z^[0 -1 ... -n].'
H(z) = ------- = ---------------------------------------------------
      DENd(z)   [DENd(1) DENd(2) ... DENd(n+1)] * z^[0 -1 ... -n].'
```

[Ad,Bd,Cd,Dd] = bilinear(Aa,Ba,Ca,Da,Fs) convert the continuous-time state-space system in matrices Aa, Ba, Ca, and Da:

```
.
x(t) = Aa*x(t) + Ba*u(t)
y(t) = Ca*x(t) + Da*u(t)
```

to the discrete-time state-space system:

```
x(n+1) = Ad*x(n) + Bd*u(n)
y(n)   = Cd*x(n) + Dd*u(n)
```

⚠ Output arguments should **NOT** be omitted, becuase they are used for input argument type differentiation.

Examples

Compatibility

See also

blackman

Blackman window

Syntax

W = blackman(N)

Definition

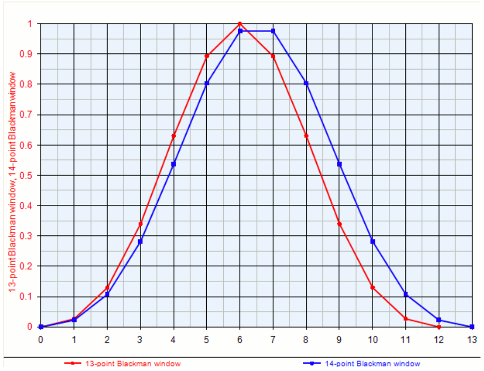
This function returns a column vector containing a Blackman window with N points, N being a positive integer greater than 2. The Blackman window is composed of raised cosine windows scaled to have a base value of 0 and an apex value of 1 as follows:

```
_blackman_value_at_n_of_N_ = 0.42 - 0.5 * cos( 2*pi*n/N ) + 0.08 * cos( 4*pi*n/N ), 0 <= n <= N
```

When N is odd, the apex is explicitly an part of the window function. When N is even, the apex is not explicitly sampled but rather the two sample points which flank the apex are represented in the returned vector.

Note
blackman(2), a redundant usage of this function returns [0 0] whereas blackman(1) returns [1].

Examples:



Compatibility
scalar

See Also
bartlett (users), *gausswin* (users), *hamming* (users), *hann* (users), *rectwin* (users)

butter

Butterworth digital and analog filter design

Syntax

```
    [B,A] = butter(N,Wc)
   [B,A] = butter(N,Wc,'ftype')
   [B,A] = butter(N,Wc,'s')
   [B,A] = butter(N,Wc,'ftype','s')
 [Z,P,K] = butter(N,Wc ...)
[A,B,C,D] = butter(N,Wc ...)
   [...] = butter(N,Wc,'ftype','fdomain')
```

Parameters

Name	Definition	Compatibility	Usage	Default	Example
order(N)	order of Butterworth filter	positive integer >= 3	required		5
normfreq(Wc)	normalized frequency or range of frequencies defining filter	normalized scalar or 2-part vector	required		0.3
ftype	type of filter	enumerated as 'low','high','pass' or 'stop'	optional	'low'	'pass'
fdomain	digital (Z-domain) or analog (S-domain) filter	'z' or 's'	optional	'z'	's'

Definition

- Depending on the list out output arguments, this function delivers a numerator-denominator or a pole-zero-gain definition of a maximally-flat Butterworth filter response. Input arguments consist of order, normalized frequency range and the optional enumerated choice of filter type.
- Full syntax: [...] = butter(N,Wc,'ftype','fdomain'). N is filter order, Wc is the 3'dB cutoff frequency. 'ftype', which is
  'low' --low pass,
  'high' --high pass,
  'pass' --band pass, or
  'stop' --band stop,
  specifying the filter type, 'low' by default. 'fdomain', which is
  's' – s-domain or analog, or
  'z' – z-domain or digital,
  specifying the filter domain, 'z' by default.
- If Wc is a scalar, the filter type may be 'low' or 'high'. For digital filter, Wc should be 0<Wc<1, where 1 corresponds to half the sample rate. For analog filter, Wc should be 0<Wc<Inf rad/s.
- If Wc is a two-element vector, Wc=[Wl,Wh] and Wl<Wh, the filter type may be 'pass' or 'stop' which shall be 2N order.
- [B,A] = BUTTER(...) returns the filter coefficients in vectors B (numerator) and A (denominator). The coefficients are in descending powers of z or s.
- [Z,P,K] = BUTTER(...) returns the zeros, poles and gain.
- [A,B,C,D] = BUTTER(...) is the state-space version.

⚠ Output arguments should **NOT** be omitted, becuase they are used for input argument type differentiation

Examples:

Note that while zeros and poles are expressed as column vector, numerator and denominator coefficients are expressed as row vectors. Gain is always expressed as a real valued scalar variable.

Formula	zeros	poles	gain	num	denom
butter(3, 0.5)	[-1+j4.714e-6; -1-j4.714e-6; -1]	[j/√3; -j/√3;0]	1/6	[1/6, 1/2, 1/2, 1/6]	[1, 0, 1/3, 0]
butter(3, 0.5, 'high')	[1+j4.714e-6; 1-j4.714e-6; 0]	[j/√3; -j/√3;0]	1/6	[1/6, -1/2, 1/2, -1/6]	[1, 0, 1/3, 0]
butter(3, [0.25,0.75], 'pass')	[1; 1+j2.597e-6; 1-j2.597e-6; -1; -1+3.772e-6; -1-j3.772e-6]	[-0.537+j0.537; -0.537-j0.537; 0.537+j0.537; 0.537+j0.537; j7.451e-9; -j7.451e-9]	1/6	[1/6, 0, -1/2, 0, 1/2, 0, -1/6]	[1, 0, 0, 0, 1/3, 0, 0]
butter(3, [0.25,0.75], 'stop')	[-3.055e-6+j; -3.055e-6-j; 3.055e6+j; 3.055e-6-j; j; -j]	[-0.537+j0.537; -0.537-j0.537;0.537+j0.537; 0.537-j0.537; 9.125e-9; 9.125e-9]	1/6	[1/6, 0, 1/2, 0, 1/2, 0, 1/6]	[1, 0, 0, 0, 1/3, 0, 0]

See Also
buttord (users),*cheby1* (users), *cheby2* (users), *ellip* (users), *filter* (users)

buttord

Butterworth filter order and cutoff frequency calculation

Syntax

[N,Wc] = buttord(Wp,Ws,Rp,Rs)

[N,Wc] = buttord(Wp,Ws,Rp,Rs,'s')

Definition

1. [N,Wc] = buttord(Wp,Ws,Rp,Rs) returns the minimum order N of a butterworth filter whose passband attenuation is less than Rs dB and stopband attenuation is at the most Rp dB. Wc, the Butterworth natural frequency (the 3'dB cutoff frequency) is also returned. Wp and Ws are passband and stopband edge frequencies, normalized by half sample frequency to (0,1) (1 corresponds to pi radians/sample). For example,

```
Lowpass:    Wp = .2,      Ws = .3
Highpass:   Wp = .4,      Ws = .3
Bandpass:   Wp = [.3 .6], Ws = [.1 .7]
Bandstop:   Wp = [.2 .8], Ws = [.3 .7]
```

[N,Wc] = buttord(Wp,Ws,Rp,Rs,'s') is the analog filter version, where Wp and Ws are in radians/second.
buttord(Wp, Ws, Rp, Rs, 'z') is the same as buttord(Wp, Ws, Rp, Rs).
Examples

Compatibility

See also
butter (users), *cheb1ord* (users), *cheb2ord* (users)

ceil

Syntax
y = ceil(x)

Definition
ceil returns the smallest integer greater than or equal to the argument. If x is complex, only the real part is used. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
ceil(10)	10
ceil(complex (1.5 , 6))	2
ceil([-0.5, 0.5])	[0 , 1]

Compatibility
Numeric scalars, vectors, arrays

See Also
floor (users)

cheb1ord

Minimum order calculation for Chebyshev Type I filter

Syntax

[N,WN] = cheb1ord(WP,WS,DBP,DBS)

[N,WN] = cheb1ord(WP,WS,DBP,DBS,'s')

Definition

- This function calculates the minimum order for Chebyshev Type I filter.
- [N,WN] = cheb1ord(WP,WS,DBP,DBS) returns the order N for digital Chebyshev filter that has no more than DBP loss in passband and at least DBS attenuation in the stop band. WP is also returned in WN.
- [N,WN] = cheb1ord(WP,WS,DBP,DBS,'s') returns the order N for analog Chebyshev filter that has no more than DBP loss in passband and at least DBS attenuation in the stop band. WP is also returned in WN.

⚠ Output arguments should **NOT** be omitted

Examples

Compatibility

See also
cheb2ord (users)

cheb2ord

Minimum order calculation for Chebyshev Type II filter

Syntax

[N,WN] = cheb2ord(WP,WS,DBP,DBS)

[N,WN] = cheb2ord(WP,WS,DBP,DBS,'s')

Definition

1. This function calculates the minimum order for Chebyshev Type I filter.
2. [N,WN] = cheb2ord(WP,WS,DBP,DBS) returns the order N for digital Chebyshev filter that has no more than DBP loss in passband and at least DBS attenuation in the stop band. WP is also returned in WN.
3. [N,WN] = cheb2ord(WP,WS,DBP,DBS,'s') returns the order N for analog Chebyshev filter that has no more than DBP loss in passband and at least DBS attenuation in the stop band. WP is also returned in WN.

   ⚠ Output arguments should **NOT** be omitted

Examples

Compatibility

See also
buttord (users), *cheb1ord* (users), *cheby2* (users)

cheby1

Chebyshev Type I filter desgin

Syntax

[num, denom] = cheby1(order, normripple, normfreq, ftype, domain)

[zeros, poles, gain] = cheby1(order, normripple, normfreq, ftype, domain)

Parameters

Name	Definition	Compatibility	Usage	Default	Example
order	order of Butterworth filter	positive integer >= 3	required		5
normripple	normalized ripple in passband	positive real	required		0.1
normfreq	normalized frequency or range of frequencies defining filter	normalized scalar or 2-part vector	required		0.3
ftype	type of filter	enumerated as 'low','high','pass' or 'stop'	optional	'low'	'pass'
domain	digital (Z-domain) or analog (S-domain) filter	'z' or 's'	optional	'z'	's'

Definition

Depending on the list out output arguments, this function delivers a numerator-denominator or a pole-zero-gain definition of a Chebyshev filter response of Type 1, which allows ripples in the passband and creates a maximally flat stopband. Input arguments consist of order, normalized in-band ripple, normalized frequency range and the optional enumerated choice of filter type.

Examples:

Note that while zeros and poles are expressed as column vector, numerator and denominator coefficients are expressed as row vectors. Gain is always expressed as a real valued scalar variable.

Formula	zeros	poles	gain	num	denom
cheby1(3, 0.1, 0.5)	[-1; -1; -1]	[-0.1885+j0.659; 0.0155; -0.1885+j0.659]	0.227	[0.227, 0.682, 0.682, 0.227]	[1, 0.361, 0.464, -0.007]
cheby1(3, 0.1, 0.5, 'high')	[1 1 1]	[0.1885+j0.659; -0.0155; 0.1885-j0.659]	0.227	[0.227, -0.682, 0.682, -0.227]	[1, -0.361, 0.464, 0.007]
cheby1(3, 0.1, [0.25,0.75], 'pass')	[1; 1; 1; -1; -1; -1]	[0.661+j0.499; j0.125; 0.661-j0.499; -0.661=j0.499; -j0.125; -0.661+j0.499]	0.227	[0.227, 0, -0.682, 0, 0.682, 0, -0.227]	[1, 0, -0.361, 0, 0.464, 0 0.007]
cheby1(3, 0.1, [0.25,0.75], 'stop')	[-j; j; -j; j; -j; j]	[0.499-j0.661; 0.125; 0.499+j0.661; -0.499+j0.661; -0.125; -0.499-j0.661]	0.227	[0.227, 0, 0.682, 0, 0.682, 0, 0.227]	[1, 0, 0.361, 0, 0.464, 0, -0.007]

See Also
butter (users), *cheb1ord* (users), *cheby2* (users), *ellip* (users)

cheby2

Chebyshev Type II filter desgin

Syntax

[num, denom] = cheby2(order, normripple, normfreq, ftype, domain)

[zeros, poles, gain] = cheby2(order, normripple, normfreq, ftype, domain)

Parameters

Name	Definition	Compatibility	Usage	Default	Example
order	order of Butterworth filter	positive integer >= 3	required		5
normripple	normalized ripple in stopband	positive real	required		0.1
normfreq	normalized frequency or range of frequencies defining filter	normalized scalar or 2-part vector	required		0.3
ftype	type of filter	enumerated as 'low','high','pass' or 'stop'	optional	'low'	'pass'
domain	digital (Z-domain) or analog (S-domain) filter	'z' or 's'	optional	'z'	's'

Definition

Depending on the list out output arguments, this function delivers a numerator-denominator or a pole-zero-gain definition of a Chebyshev filter response of Type 2, which allows ripples in the stopband and creates a maximally flat passband. Input arguments consist of order, normalized out-of-band ripple, normalized frequency range and the optional enumerated choice of filter type.

Examples:

Note that while zeros and poles are expressed as column vector, numerator and denominator coefficients are expressed as row vectors. Gain is always expressed as a real valued scalar variable.

Formula	zeros	poles	gain	num	denom
cheby2(3, 0.1, 0.5)	[-0.143+j0.990; -0.143-j0.990; -1]	[-0.138-j0.962; -0.903; -0.137+j0.962]	0.924	[0.924, 1.188, 1.188, 0.924]	[1, 1.178, 1.192, 0.853]
cheby2(3, 0.1, 0.5, 'high')	[0.143-j0.990; 0.143+j0.990; 1]	[0.137+j0.962; 0.903; 0.137-j0.962]	0.924	[0.924, -1.188, 1.188, -0.924]	[1, -1.178, 1.192, -0.853]
cheby2(3, 0.1, [0.25,0.75], 'pass')	[-0.756+j0.655; 0.756+j0.655; 0.756-j0.655; -0.756-0.655; 1; -1]	[0.745+j0.646; 0.951; 0.745-j0.646; -0.745-j0.646; -0.951; -0.745+j0.646]	0.924	[0.924, 0, -1.188, 0, 1.188, 0, -0.924]	[1, 0, -1.178, 0, 1,192, 0, -0.853]
cheby2(3, 0.1, [0.25,0.75], 'stop')	[0.655+j0.756; -0.655+j0.756; -0.655-j0.756; 0.655-j0.756; -j; j]	[0.646-j0.745;-j0.951; 0.646+j0.745; -0.646+j0.745; j0.951; -0.646-j0.745]	0.924	[0.924, 0, 1.188, 0, 1.188, 0, 0.924]	[1, 0, 1.178, 0, 1.192, 0, 0.853]

See Also
butter (users), *cheb1ord* (users), *cheby1* (users), *ellip* (users)

class

Syntax
type = class(object)

Definition
This function returns the type of class of the supplied *object* as a string *type*. The input argument is evaluated as an expression so a combination of existing objects can be applied to this parameter.

Example

- **char**

```
c1 = class( ['This','is','a','char','class','vector'] )
% This is a vector of strings or an array of characters
% c1 = 'char'
```

- **cell**

```
c2 = class( {'This','is','a','char','class','vector'} )
% This is a cell array of characters
% c2 = 'cell'
```

- **double**

```
c3a = class( [1 2 3; 4.5 5 6] )
% This is an array of double precision floating point numbers
% c3a = 'double'
% Note real-valued integers and floating point assigned the 'double'
% whereas,
c3b = class( 4+5i )
% complex-valued numbers are assigned 'complex double'
% c3a = 'complex double'
```

- **logical**

```
c4 = class( [ 3<=4, length('were') < size([2,1])] )
% The 1st expression is evaluating whether or not 3<=4.
% The 2nd expression is evaluating wthere the length of
% the string 'were' is greater than the length of the supplied
% numeric vector [2,1]. Both are logical expressions, making
% the supplied vector of logical class.
% c4 = 'logical'
```

- **struct**

```
c5 = class( struct('Name',{'FirstName','LastName'},'Date Of Birth', [23 04 1999]) )
% The expression defines a structure, thus
% c5 = 'struct'
```

Compatibility
all

See Also
struct (users)

conj

Syntax
y = conj(x)

Definition
conj returns the complex conjugate of the argument. The conjugate of a complex number x + jy is x - jy. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
conj(1+2j)	1 - 2j
conj([1 + 2j, 3 - 4j])	[1 - 2j, 3 + 4j]

Compatibility
Numeric Scalars, Arrays, Vectors

conv

Convolution of u and v

Syntax

y = conv(x1, x2)

Definition

This function performs the algebraic convolution between the two vector valued inputs *x1* and *x2*. Given the lengths of the vectors to be *lN* = length(*xN*), N = {1, 2}, the result is of length equal to sum of all lengths minus 1.

Examples:

```
a = [ 2, 3 ]
b = [ 5, 6, 7 ]
c = conv( a, b )
% c = [10 27 32 21] because
% c(1) = a(1)*b(1) = 10
% c(2) = a(1)*b(2) + a(2)*b(1) = 12 + 15 = 27
% c(3) = a(1)*b(3) + a(2)*b(2) = 14 + 18 = 32
% c(4) = a(2)*b(3) = 21
```

Compatibility
Real and complex valued scalars and vectors. Multi-dimensional arrays are not supported.

See Also
filter (users), *fft* (users), *ifft* (users)

convdeintrlv

Permute data with specified shift register group

Syntax

Y = convdeintrlv(X,FIFONum,Delta)

Y = convdeintrlv(X,FIFONum,Delta,InitState)

[Y,FinalState] = convdeintrlv(X,FIFONum,Delta,...)

Definition

Y = convdeintrlv(X,FIFONum,Delta) restores ordering the data in X with shiftregister (FIFO) group. The i'th FIFO can hold (FIFONum-i)*Delta data,i=1,2,...,FIFONum. FIFONum and Delta should be the same as that in convintrlv.

Y = convdeintrlv(X,FIFONum,Delta,InitState) initialize the shift registersspecified in InitState instead of all zeros.

[Y,FinalState] = convdeintrlv(X,FIFONum,Delta,...) returns final state ofshift registers in FinalState which may be used as initial state of the nextprocess when dealing with consecutive data.

convdeintrlv is implemented by calling function *muxintrlv* (users).

Examples

Compatibility

See also
convintrlv (users), *muxdeintrlv* (users).

convenc

Convolutionally encode binary data

Syntax

cBits = convenc(uBits,TRELLIS)

cBits = convenc(uBits,TRELLIS,puncPat)

cBits = convenc(uBits,TRELLIS,puncPat,initState)

[cBits, finalState] = convenc(...)

Definition

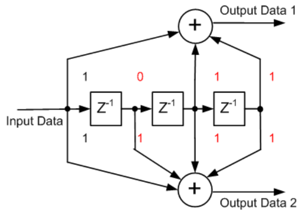
cBits = convenc(uBits,TRELLIS) encodes the binary vector uBits(uncoded bits) with the struct TRELLIS generated with function *poly2trellis* (users). The output cBitsis the coded bits with the length of length(uBits)*log2(trellis.numOutputSymbols)*log2(trellis.numInputSymbols).

cBits = convenc(uBits,TRELLIS,puncPat) use puncture pattern vector puncPatto delete specified bits after trellis encoding to get a higher coding raterelative to the mother code (before puncturing). puncPat is a vectorconsists of 1's and 0's, where 1 means reserve a bit and 0 meansdelete a bit. puncPat may be set [] or a vector with more than 2 elements,which means a scalar is illegal.

cBits = convenc(uBits,TRELLIS,puncPat,initState) allows the encoder to use initState scalar, defaults to 0, as initial state of inner registers.initState must be the last argument and in the range of[0,TRELLIS.numStates-1]. puncPat here may be omitted or set [].

[cBits, finalState] = convenc(...) regurns the final state of innerregisters inside the encoder, which is useful for consecutive processingin case uBits is very long.

Examples
Encoding for the (2,1,3) code shown below,




```
ConsLen = [4]; % there're 3 registers in the encoder
CodeGen = [13,17]; % octal [1 0 1 1] and [1 1 1 1];
trellis = poly2trellis(ConsLen, CodeGen);
uBits = [1 0 0 1 1 0 1 0 1 1 0 0 0 1 1 0 1 0 0 0]; % 3 tail 0's for return coder state to zero,
length is 20
cBits = convenc(uBits,trellis); % encoding, output length is 40
% puncture the code to 2/3 code rate
puncPat = [1 1 1 0];
cBits_punc = convenc(uBits,trellis,puncPat); % encoding and puncture, output length is 30
% same as
% cBits_punc = reshape(cBits,4,10);
% cBits_punc = cBits_punc(find(puncPat==1),:);
% cBits_punc = cBits_punc(:)';
% continuous encoding
initState = 0;
cBits_Cont = zeros(1,numel(uBits)*2);
blkSize = 5;
blkNum = numel(uBits)/blkSize;
for blk = 0:blkNum-1
   [cBitsBlk,initState] = convenc(uBits(blkSize*blk+(1:blkSize)),trellis,[],initState);
   cBits_Cont(blkSize*2*blk+(1:blkSize*2)) = cBitsBlk;
end
```

results

```
cBits =
 1 1 0 1 1 1 0 0 1 0 1 0 1 1 1 0 0 0 0 1 1 0 0 0 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 1
cBits_punc =
 1 1 0 1 1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1 1 1 0 1 1 1 1 1 1 1
cBits_Cont =
 1 1 0 1 1 1 0 0 1 0 1 0 1 1 1 0 0 0 0 1 1 0 0 0 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 1
```

Compatibility

See also
vitdec (users), *poly2trellis* (users)

convintrlv

Permute data with specified shift register group

Syntax

Y = convintrlv(X,FIFONum,Delta)

Y = convintrlv(X,FIFONum,Delta,InitState)

[Y,FinalState] = convintrlv(X,FIFONum,Delta,...)

Definition

Y = convintrlv(X,FIFONum,Delta) rearranges the data in X with shift register(FIFO) group. The i'th FIFO can hold (i-1)*Delta data, i=1,2,...,FIFONum.The input data is fed into the shift registers, from the first to the last,in sequence and periodicly. Assuming X={x1,x2,x3,...}, x1 is fed intobranch 1, x2 is fed into branch 2, ... The output picks up data from outputof each shift register, from the first to the last, in sequence andperiodicly. Note that the data feeding to each shift register and datapicking up from each shift register are synchronou. If X is a matrix, eachcolumn is treated as an independent signal. All shift registers areinitialized with zeros before process begins.

Y = convintrlv(X,FIFONum,Delta,InitState) initialize the shift registersspecified in InitState instead of all zeros. InitState is a structurecomposed of variables InitState.value and InitState.index. InitState.valuehas the same number of columns as X, each stores the initial state of shiftregisters (from first to last). FinalState.index represents the index of theshift register into which the first symbol shall be fed. Assuming FIFONum is4, InitState.value=[1 2 3 4 5 6].', InitState.index=2, then we haveinitial state:

[] --FIFO 1
[1] --FIFO 2
[2 3] --FIFO 3
[4 5 6], --FIFO 4

and shall start processing from the FIFO 2.

[Y,FinalState] = convintrlv(X,FIFONum,Delta,...) returns final state of shiftregisters in FinalState which may be used as initial state of the nextprocess when dealing with consecutive data. FinalState is a struct composedof variables FinalState.value and FinalState.index. FinalState.value has thesame number of columns as X, each stores the final state of shift registers(from first to last) after processing the corresponding column of X.FinalState.index represents the index of the shift register from which thenext consecutive processing shall begin.

convintrlv is implemented by calling function *muxintrlv* (users).

Examples

Compatibility

See also
convdeintrlv (users), *muxintrlv* (users)

cos

Syntax
y = cos(x)

Definition
cos returns the cosine of a radian-valued argument. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
cos(0)	1
cos(pi)	-1
cos(pi / 2)	0
cos(pi / 4)	0.707
cos([2*pi/3; pi/2])	[-0.5; 0]

Compatibility
Numeric scalars, Vectors, Arrays

See Also
cosd (users)

sin (users)
tan (users)

cosd

Syntax
y = cosd(x)

Definition
cosd returns the cosine of a degree-valued argument. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
cosd(0)	1
cosd(180)	-1
cosd(90)	0
cosd(45)	0.707
cosd([60; 90])	[-0.5; 0]

Compatibility
Numeric scalars, Vectors, Arrays

See Also
cos (users)

cosh

Syntax
y = cosh(x)

Definition
cosh returns the hyperbolic cosine of the argument, equivalent to (exp(x) + exp(-x)) / 2. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
cosh(1)	1.543
cosh(pi / 3)	1.6
cosh([pi/6; 0])	[1.14; 1]

Compatibility
Numeric scalars, Vectors, Arrays

See Also
acosh (users)

cot

Syntax
y = cot(x)

Definition
cot returns the cotangent of a radian-valued argument, which is equivalent to 1 / tan(x). This function operates on an part-by-part basis on arrays.

Compatibility
Numeric scalars, Vectors, Arrays

cotd

Syntax
y = cotd(x)

Definition
cotd returns the cotangent of a degree-valued argument. This function operates on an part-by-part basis on arrays.

Compatibility
Numeric scalars, Vectors, Arrays

coth

Syntax
y = coth(x)

Definition
coth returns the hyperbolic cotangent of the argument. This function operates on an part-by-part basis on arrays.

Compatibility
Numeric scalars, Vectors, Arrays

crcdec

Cyclic redundancy check decoder

Syntax

[msg, errFlag, syndrome] = crcdec(code, genPoly, initState)

Definition

- **code:** input message to be decoded (checked)
- **genPoly:** generation polynomial of CRC code, binary vector, highest degree first. If g(x)=x^3+x+1, then genPoly=[1 0 1 1]
- **initState:** initial state of registers in CRC decoder, highest degree first i.e. initState=[D(N-K-1), D(N-K-2), ..., D(1), D(0)], where N and K are codeword length and message length. Default value is all zeros.
- **msg:** output message (discarding parity from code, no error correction)
- **errFlag:** error flag, 1 means there are errors in code
- **syndrome:** checksum of code, equals to the CRC parity of first K bits of code XOR the last N-K bits of code

Examples

Cyclic (7,4) Hamming code, g(x)=x^3+x+1, i.e. genPoly=[1 0 1 1]

```
code -----> + <--D(2)<-------D(1)<-- + <--D(0)<--
code(1)     |                        ^          |
first       |                        |          |
           |                        |          |
             --->--------------------------------
```

msg = code(1:K);

Compatibility

See also
crcenc (users)

crcenc

Cyclic redundancy check encoder

Syntax

[code, parity] = crcenc(msg, genPoly, initState)

Definition

- **msg** : input message to be encoded
- **genPoly** : generation polynomial of CRC code, binary vector, highest degree first. If g(x)=x^3+x+1, then genPoly=[1 0 1 1]
- **initState** : initial state of registers in CRC code, highest degree first i.e. initState=[D(N-K-1), D(N-K-2), ..., D(1), D(0)], where N and K are codeword length and message length. Default value is all zeros.
- **code** : msg appended by parity
- **parity** : checksum of input message

Examples

Cyclic (7,4) Hamming code, g(x)=x^3+x+1, i.e. genPoly=[1 0 1 1]

```
msg -----> + <--D(2)<-------D(1)<-- + <--D(0)<--
msg(1)     |                        ^          |
first      |                        |          |
          |                        |          |
           --->--------------------------------
```

parity = [D(2), D(1), D(0)]

code = [msg, parity]

Compatibility

See also
crcdec (users)

csc

Syntax
y = csc(x)

Definition
csc returns the cosecant of a radian-valued argument. This function operates on an part-by-part basis on arrays.

Compatibility
Numeric scalars, Vectors, Arrays

cscd

Syntax
y = cscd(x)

Definition
cscd returns the cosecant of a degree-valued argument. This function operates on an part-by-part basis on arrays.

Compatibility
Numeric scalars, Vectors, Arrays

csch

Syntax
y = csch(x)

Definition
csch returns the hyperbolic cosecant of the argument. This function operates on an part-by-part basis on arrays.

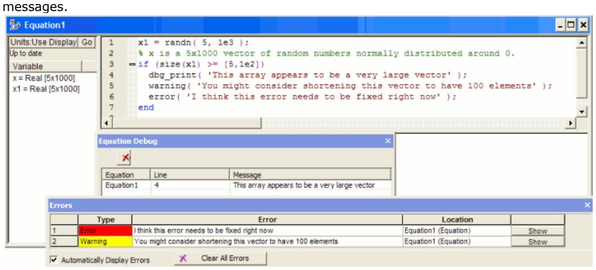
Compatibility
Numeric scalars, Vectors, Arrays

dbg_print

Syntax
dbg_print('message')

Definition
This function can be invoked from within a set of equations on an equations page in order to report execution status to the **Equation Debug** window. Note that the window for debugging equations is not the same as the **Error Log**. The debug window can be invoked by selecting **View > Advanced Windows > Equation Debug**.

Examples:
Note the difference between the reporting windows and formats for debug and non-debug messages.



Compatibility
string

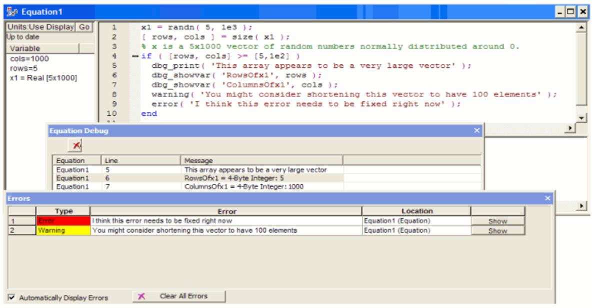
See Also
error (users)
dbg_showvar (users)
warning (users)

dbg_showvar

Syntax
dbg_showvar(name, variable)

Definition
This function can be invoked from within a set of equations on an equations page in order to report the current value of a *variable* to the **Equation Debug** window by the supplied *name*. Note that the window for debugging equations is not the same as the **Error Log**. The debug window can be invoked by selecting **View > Advanced Windows > Equation Debug**.

Examples:
In the following example the use model of dbg_showvar() is shown along side that of other relevant Mathematical Language functions.



Formula	Message in Equation Debug
dbg_showvar('Expression', 2+3);	'Expression = 8-byte Real: 5'
string1 = 'hello world'; dbg_showvar('Greeting', string1);	'Greeting = Array[1x11] of type Char: hello world'
vector1 = [1 2 3]; dbg_showvar('Vector', vector1);	'Vector = Array[1x3] of type 8-Byte Real: 1 2 3'
array1 = [1 2; 3+2j, 9]; dbg_showvar('Array', array1);	'Array = Array[2x2] of type 16-Byte Complex: 1 2 3+2i, 9'
cell1 = {'This','is','a','sentence','.'}; dbg_showvar('Cell', cell1);	'Cell = Array[1x5] of type Variant: [1x4 char] [1x2 char] ['a'] [1x8 char] ['.']'
struct1 = struct('name',{'Jane','Doe'},'AgE', 37); dbg_showvar('Struct', struct1);	'Struct = Array[1x2] of type Object:[1x2\ struct] with fields: name AgE'

Compatibility
name - string
variable - any pre-defined variable or expression

See Also
error (users)
dbg_print (users)
warning (users)

de2bi

Convert decimal numbers to binary vectors

Syntax

b = de2bi(d)

b = de2bi(d,n)

b = de2bi(d,n,p)

b = de2bi(d,[],p)

b = de2bi(d,flg)

b = de2bi(d,n,flg)

b = de2bi(d,n,p,flg)

b = de2bi(d,[],p,flg)

Definition

B = de2bi(D) converts positive decimal integer to binary row vector. If D isMB-NB matrix,B should be a MB*NB-N matrix where N is specified either by paramN or P

B = de2bi(D,N), N specifies the column of B. if N is smaller than the elementsin D actually need, there is an error.

B = de2bi(D,FLG), FLG can be 'left-msb' or 'right-msb'. default is 'right-msb'.

B = de2bi(D,N,P) converts positive decimal integer to base-P row vector. If N issmaller than elements in B actually need, there is an error.

B = bi2de(D,[],P) means the column of B is specified by P.

Examples

Compatibility

See also
bi2de (users)

dec2hex

Decimal to hexadecimal number string conversion

Syntax:
dec2hex(number[,places])

Definition:
dec2hex function converts a decimal number to a hexadecimal number string. Places is an optional field, specifying to zero pad to that number of spaces. If places is too small or negative #NUM! error is returned.

Examples:
dec2hex(42) equals 2A.

See Also:
hex2dec (users)

deconv

Syntax
[a,b] = deconv(c,d)

Definition
[a,b] = deconv(c,d) deconvolves a vector d out of a vector c and returns it in vector a, and the remainder in b so that c = conv(d,a) + b.

If vectors c and d contain the coefficients of a polynomial, then convolving them is equivalent to mutiplying the polynomials, and deconvolving is equivalent to dividing the polynomials.

Examples:

Formula	Result
b = [1 2 3 4] a = [10 20 30] [q,r] = deconv(a,b)	q = [10 20 30] r = [0 0 0 0 0 0]

Compatibility
vector

See Also

conv (users)

deintrlv

Reorder data back with specified permutation table

Syntax

Y = deintrlv(X,PermTab)

Definition

Y = deintrlv(X,PermTab) rearranges the data in X with indices given in PermTab as a inverse process of INTRLV. If X is a vector of length N, length of PermTab must be a factor of N, i.e. mod(N,length(PermTab))==0. If X is a matrix, PermTab must be a factor of the number of rows of X, and each column of X is treated as an independent signal.

Examples

```
b = deintrlv([10 40 20 50 30 60; 70 100 80 110 90 120].', [1 4 2 5 3 6])
b =
10 70
20 80
30 90
40 100
50 110
60 120
```

Compatibility

depuncture

Restores erasures based on puncture pattern

Syntax

Y = depuncture(X, puncPat)

Y = depuncture(X, puncPat, stuffVal)

Definition

- **puncPat** : a vector of 1's and 0's, such as [1 0 1 1]
- **stuffVal** : stuff values to be filled in restored position, 0 by default.

Examples

```
x = [1 3 4 5 7 8 9];
puncPat = [1 0 1 1];
y = puncture(x,puncPat),
y =
 1 0 3 4 5 0 7 8 9 0
```

Compatibility

See also
puncture (users)

diag

Syntax
V = diag(x [, a])
v = diag(X)

Definition
If x is a vector, diag(x) gives a matrices V with x on main diagonal. diag(x, a) returns an abs(a)+n (if there are n parts in x) square matrix with the parts of a on the a-th diagonal, main diagonal when a = 0, upper diagonal when a>0, and lower diagonal when a<0.
If X is a matrix, diag(X) returns its main diagonals to a column vector v.

Examples:

Formula	Result
diag([2,3])	[2, 0; 0, 3]
diag([1,5], 1)	[0, 1, 0; 0, 0, 5; 0, 0, 0]
diag([1 2 3; 4 5 6; 7 8 9])	[1; 5; 9]

Compatibility
Numeric vectors, Vectors, Matrices

diff

Syntax
A = diff(B)
A = diff(B,r)
A = diff(B,r,dim)

Definition
text here
A = diff(B) returns, in the vector A, the difference between each part in B.
A = diff(B,r) recurses the diff function r times, to find the rth difference.
A = diff(B,r,dim) recurses the diff function r times, to find the rth difference in the scalar dimension dim. If r>= dim, then an empty array is returned.

The dim argument is optional and specifies which dimension to operate along. For example, if dim is 1, this function operates on each column of the argument. If the argument is omitted, the first non-singleton dimension is chosen as the dimension to operate along.

Examples:

Formula	Result
B = [1 5 15 35] A = diff(B)	[4 10 20]
N = diff(A)	[6 10]
Z = diff(A,2)	[4]

Compatibility
scalar, vector, array

See Also

sum (users)

downsample

Downsample input signal

Syntax

Y = downsample(X,R)

Y = downsample(X,R,OFFSET)

Definition

1. Y = downsample(X,R) downsamples input signal X by keeping the first of every R continuous samples. X may be a vector or a matrix (one signal per column). For matrix, downsampling is applied on each column respectively.
2. Y = downsample(X,R,OFFSET) specifies an optional sample offset. OFFSET should be an integer within [0,R-1] and is 0 by default.

Examples

```
x = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20].';
y = downsample(x, 4);
z = upsample(x, 4, 1);
p = [1 5 9 13 17].'; % y equals to p
q = [2 6 10 14 18].'; % z equals to q
```

Compatibility

See also
upsample (users), *upfirdn* (users), *interp* (users), *resample* (users)

dpskdemod

Differential phase-shift keying demodulation

Syntax

Definition

Examples

Compatibility

No

See also
dpskmod (users)

dpskmod

Differential phase-shift keying modulation

Syntax

Definition

Examples

Compatibility

No

See also
dpskdemod (users)

eig

Syntax
X = eig(Y)
X = eig(Y,Z)
[U,X] = eig(Y)
[U,X] = eig(Y,Z)
[U,X] = eig(Y,Z,flag)

Definition
X = eig(Y) returns, in vector X, the eigenvalues of the matrix Y.

X = eig(Y,Z) returns, in vector X, the generalized eigenvalues, as long as Y and Z are square matrices.

[U,X] = eig(Y) produces matrices containing the eigenvalues in X, and eigenvectors in U, so: Y*U = U*X

[U,X] = eig(Y,Z) produces a diagonal matrix, X, that contains the generalized eigenvalues, and a full matrix, U, containing the eigenvectors in columns, so: Y*U = Z*U*X

[U,X] = eig(Y,Z,flag) produces the eigenvalues and eigenvectors using a specified algorithm, flag:
'chol' - Computes using Cholesky factorization of Z.
'qz' - Computes using QZ algorithm.

Examples:

Formula	Result
Z =[3 -2 -.9 2*eps; -2 4 1 -eps; -eps/4 eps/2 -1 0; -.5 -.5 .1 1] [VZ,DZ] = eig(Z) [VY,DY] = eig(Z,'nobalance')	Z*VZ - VZ*DZ Z*VY - VY*DY

ellip

Syntax
[num, denom] = ellip(order, passnormripple, stopnormripple, normfreq, ftype, domain)
or
[zeros, poles, gain] = ellip(order, passnormripple, stopnormripple, normfreq, ftype, domain)

Parameters

Name	Definition	Compatibility	Usage	Default	Example
order	order of Butterworth filter	positive integer >= 3	required		5
passnormripple	normalized ripple in passband	positive real	required		0.1
stopnormripple	normalized ripple in stopband	positive real	required		0.1
normfreq	normalized frequency or range of frequencies defining filter	normalized scalar or 2-part vector	required		0.3
ftype	type of filter	enumerated as 'low','high','pass' or 'stop'	optional	'low'	'pass'
domain	digital (Z-domain) or analog (S-domain) filter	'z' or 's'	optional	'z'	's'

Definition
Depending on the list out output arguments, this function delivers a numerator-denominator or a pole-zero-gain definition of an elliptic filter response, which allows controlled amounts of ripples both in the pass and stop bands. Input arguments consist of order, normalized in- and out-of-band ripples, normalized frequency range and the optional enumerated choice of filter type.

Examples:
Note that while zeros and poles are expressed as column vector, numerator and denominator coefficients are expressed as row vectors. Gain is always expressed as a real valued scalar variable.

Formula	zeros	poles	gain	num	denom
ellip(3, 0.1, 0.1, 0.5)	[j; -j; -1]	[j; -j; -0.040]	0.520	[0.520, 0.520, 0.520, 0.520]	[1, 0.040, 1, 0.040]
ellip(3, 0.1, 0.1, 0.5, 'high')	[-j; j; 1]	[-j; j; 0.040]	0.520	[0.520, -0.520, 0.520, -0.520]	[1, -0.040, 1, -0.040]
ellip(3, 0.1, 0.1, [0.25,0.75], 'pass')	[-1/√2+j/√2; 1/√2+j/√2; 1/√2-j/√2; -1/√2-j/√2; 1; -1]	[1/√2+j/√2; 1/√2+j/√2; 0.2; -1/√2+j/√2; -1/√2-j/√2; -0.2]	0.520	[0.520, 0, -0.520, 0, 0.520, 0, -0.520]	[1, 0, -0.040, 0, 1, 0, -0.040]
ellip(3, 0.1, 0.1, [0.25,0.75], 'stop')	[1/√2+j/√2; -1/√2+j/√2; -1/√2-j/√2; 1/√2-j/√2; -j; j]	[1/√2+j/√2; 1/√2-j/√2; j0.2; -1/√2-j/√2; -1/√2+j/√2; -j0.2]	0.520	[0.520, 0, 0.520, 0, 0.520, 0, 0.520]	[1, 0, 0.040, 0, 1, 0, 0.040]

See Also
cheby1 (users)
butter (users)

cheby2 (users)

equalize

Equalize signal using Equalizer

Syntax

Definition

Examples

Compatibility

No

See also

erf

Syntax
y = erf(x)

Definition
This function computes the error function of each part of x.
The parts of x must be real.

Examples:

Formula	Result
erf(-1.5)	-0.9661
erf(2)	0.9953
erf([-1; -2; 1.1])	[-0.8427; -0.9953; 0.8802]

Compatibility
Real valued scalars, vectors, arrays

See Also
erfc (users)

erfc

Syntax
y = erfc(x)

Definition
This function computes the complementary error function of each part of x.
The parts of x must be real.

Examples:

Formula	Result
erfc(-1.5)	1.9661
erfc(2)	0.0047
erfc([-1; -2; 1.1])	[1.8427; 1.9953; 0.1198]

Compatibility
Real valued scalars, vectors, arrays

See Also
erf (users)

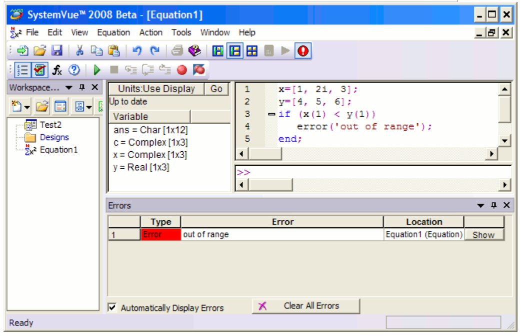
error

Syntax
error('message')

Definition
Posts the error message to the error log and also places the red error symbol on the menu button.

Examples:

Formula	Result
error('out of range')	the message "out of range" is posted to the **Error Log** as an error



Compatibility
Strings

See Also
warning (users)

exist

Syntax
y = exist(Name, Kind, Scope)

Definition
This function checks the existence of a variable or a built-in function. The Name, Kind, and Scope arguments must be strings. Kind and Scope are optional arguments, whereas Name is mandatory. The value of Name must be the name of a variable or built-in function. The exist functions returns 1 if Name is a variable in the Scope, and 5 if it is builtin function, and 0 if the specified Name is not found in the Scope.

If Kind is specified then only that kind is searched for existence. The supported values for Kind are 'var' and 'builtin'.

Default value for an Scope is 'global', a Scope argument can only be specified if Kind = 'var'. The Scope can be either a 'global', a 'local' or the name of a dataset.

Examples:

Formula	Result
iCode = exist('x')	set the variable iCode to 1 if 'x' is a variable name in global scope
iCode = exist('sin')	set the variable iCode to 5 because 'sin' is a built-in function
iCode = exist('sin','var')	set the variable iCode to 0 because 'sin' is a built-in function but it is not of Kind 'var'
iCode = exist('x','builtin')	set the variable iCode to 0 even if the variable named 'x' exist as it is not a built-in function
iCode = exist('S1','var','Design1_Data')	set the variable iCode to 1 if S1 is a variable present in dataset 'Desing1_Data'

Compatibility
Name, Kind, and Scope are strings.

See Also
getvariable (users)
setvariable (users)

exp

Syntax
y = exp(x)

Definition
This function returns the exponential of the argument. The exponential function calculates e to the power of x, where e = 2.7182817... This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
exp(1)	2.718
exp([0 , 1.5])	[1 , 4.482]
exp([-0.5 , 0.5 ; -2 , 2])	[0.607 , 1.649 ; 0.135 , 7.389]

Compatibility
Numeric scalars, Vectors, Arrays. Real and Complex.

eye

Syntax
y = eye(n)
y = eye(m, n)
y = eye(size(A))

Definition
Y = eye(n) returns the n-by-n identity matrix.

Y = eye(m, n) or eye([m n]) returns an m-by-n matrix with 1's on the diagonal and 0's elsewhere.

Y = eye(size(A)) returns an identity matrix the same size as A.

Examples
X = eye(4, 5);

eyediag

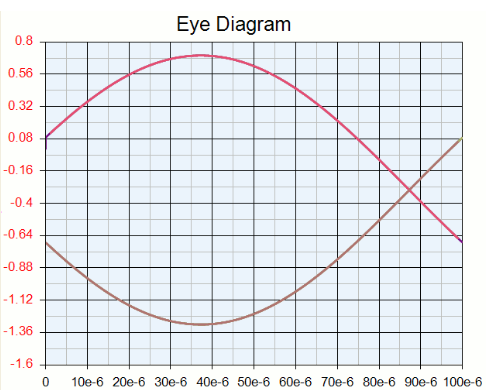
Syntax
y = eyediag(x, symbolRate, numCycles, startupDelay)

Definition
This function builds an eye diagram from a time sequence x.

Parameter	Comment	Unit	Requirement	Compatibility	Default
x	one-dimensional time sequence waveform	V	required	real-valued	
symbolRate	rate of input sequence	Hz	required	real > 0.0	
numCycles	number of unit intervals to be plotted >= 1		optional	integer >=1	1
startupDelay	number of samples that will be removed from beginning of time sequence before plotting >= 0		optional	integer > 0	0

Examples:
y = eyediag(x, 2*5e3, 1, 23)

Note that the following eye diagram was derived from a sinusoid at 5 kHz, so the unit interval was half of the 200 usec period, or just 100 usec. The data-rate or symbol-rate of this simple waveform is therefore 1/unit interval or 10 kHz. The eye-diagram itself was delayed by 23 samples to demonstrate the time-shift propert of this function.



fclose

Syntax
fclose(fileP)

Definition
This function closes the file stream referenced by *fileP* and returns a 0 if the operation is successful.

Examples:

```
fileP = fopen( 'MyFile.txt','r' );
%
% Access first 200 contiguously located floating point numbers
a = fscanf( fileP, '%f', 200 );
%
% Close file
fclose( fileP );
%
```

See Also

fgets (users)
fopen (users)
fread (users)
fprintf (users)
fscanf (users)
fwrite (users)
tcpip (users)

fft

Syntax
V = fft(X)
V = fft(X,n)
V = fft(X,[],c)
V = fft(X,n,c)

Definition
Discrete Fourier Transform (DFT) of data. Computed with FFT algorithm when possible.
The parameter len is the FFT length and is optional.
fft(X) gives the discrete Fourier transform of vector X.
fft(X,n) returns the n-point DFT. X adds zeros if n>length of X, X is truncated if n<length of X.
fft(X, [], c) gives the FFT on the dimension c.

Examples:

The following example generates a signal consisting of the sum of two sinusoids: one at 400 Hz, and one at 1500 Hz. The fft function is then used to compute the spectrum of the signal.

```
fft_len = 1024                            % length of the FFT
fs = 8000                                      % 8000 Hz sampling rate
T = 1/fs                                       % sample time
L = 1000                                       % length of signal
t = (0:(L-1))*T                                % time vector
                                              % x will be the sum of two sinusoids:
                                              % one at 400 Hz and one at 1500 Hz
x = 0.5*cos(2*pi*400*t) + cos(2*pi*1500*t)
X = fft(x, fft_len)                            % spectrum of x
X = X(1:(fft_len/2))                           % we only care about single side\-band  (the rest is
redundant)
f = fs/2 * (0:(2/fft_len):1)
```

The following graph displays the magnitude of X, the spectrum of x.

Compatibility
Vectors, Arrays, Dataset

See Also
ifft (users)

fftfilt

FFT-based FIR filtering using overlap-add method

Syntax

Y = fftfilt(B,X)

Y = fftfilt(B,X,N)

Definition

1. Y = fftfilt(B,X) filters X with the FIR filter B where FFT size is automatically selected.
2. Y = fftfilt(B,X,N) uses a FFT of at least N points.
3. If X is a matrix and B is a vector, FFTFILT filters each column of X with B and returns a matrix with the same number of columns as X.
   If X is a matrix and B is a matrix, FFTFILT filters each column of X with the corresponding column of B.
   If X is a vector and B is a matrix, FFTFILT filters X with each column of B respectively, the result is the same as that when X is a matrix with the same number of columns as B and all columns are the same.
4. FILTER performs length(B) points of multiplications for each sample. FFTFILT performs N*log2(N)/2 + N + N*log2(N)/2 or N*(1+log2(N) points of complex multiplication for every N-length(B)+1 samples. For complex X and complex B, the cost ratio of FFTFILT to FILTER is approximately (1+log2(N))*N/(N-length(B)+1)/length(B). By default, N is selected a value that minimize the ratio.

Examples

Compatibility

See also
conv (users), *filter* (users)

fgets

Syntax
y = fgets(fileP)
y = fgets(fileP, maxChars)

Definition
This function gets the next line from an open file and presents it in a string, including the newline character.

Use the argument *maxChars* to specify the maximum number of character to read. At most_maxChars_ characters will be returned.

Compatibility
fileP - pointer to an open file that is ready for reading
maxChars - maximum number of characters to be read from the next line

See Also
fclose (users)
fopen (users)
fread (users)
fprintf (users)
fscanf (users)
fwrite (users)
tcpip (users)

filter

Filter data with recursive (IIR) or nonrecursive (FIR) filter

Syntax

y = filter(b,a,X)

[y,zf] = filter(b,a,X)

Definition

1. y = filter(b,a,X)filters the data in vector/matrix X with the filter described by numerator coefficient vector b and denominator coefficient vector a.
2. [y,zf] = filter(b,a,X) returns the final conditions, zf, of the filter delays. If X is a row or column vector, output zf is a column vector of max(length(a),length(b))-1.

Examples:

```
X = [1:0.4:6]';
windowSize = 4;
Y = filter(ones(1,windowSize)/windowSize,1,X),
```

results

```
Y =
 0.25
 0.6
 1.05
 1.6
 2
 2.4
 2.8
 3.2
 3.6
 4
 4.4
 4.8
 5.2
```

Compatibility
Vectors, Matrices

See Also

find

Syntax
i = find(A)
I = find(A, n)
I = find(A, n, 'first')
I = find(A, n, 'last')
[r,c] = find(A, ...)
[r,c,x] = find(A, ...)

Definition
i = find(A) returns the indices of all the nonzero parts in array A and places them in vector i.
I = find(A n) returns an n number of indices of all the nonzero parts in an array A and places them in vector i. adding a 'first' argument means that it returns the first n indices of all the nonzero parts, and adding a 'last' argument means that it returns the last n indices.

[r,c] = find(A, ...) finds all the nonzero parts in array A and returns the row location, in r, and column location, in c.
[r,c,x] = find(A, ...) finds all the nonzero parts in array A and returns the row location, in r, and column location, in c, as well as returning the nonzero parts in a vector, x.

Examples:

Formula	Result
find([1, 0, 2, 0, 3, 5])	[1, 3, 5, 6]
find([1, 0, 2; 0, 3, 5], 3)	[1, 4, 5]
find([1, 0, 2; 0, 3, 5], 2, 'last')	[5,6]

Compatibility
Numeric arrays

finddelay

Estimate delay(s) between signals

Syntax

D = finddelay(X,Y)

D = finddelay(X,Y,MAXLAG)

Definition

D = finddelay(X,Y) returns delay D between X and Y, where X is used as reference signal. X and Y should have the same columns or at least one should be column vector. For example, If X is MX-NX matrix and Y is MY-NX matrix, D is 1-NX vector.If X is MX-NX matrix and Y is MY-1 vector, or X is MX-1 vector and Y is MY-NY matrix, D is 1-NX or 1-NY vector. The delay is estimated via normalized corrValueelation between X and Y. The result can be positive or negative. If MAXLAG is not specified, the delay should fall in the range of [-max(MX,MY)+1, max(MX,MY)-1]. when there are several delays are possible, the smallest positive delay is returned.

D = finddelay(X,Y,MAXLAG), the delay should fall in the range of [-MAXLAG(j), MAXLAG(j)] for the jth column of X or Y. MAXLAG should be row vector and the length should equal to the larger column of X and Y. MAXLAG should fall in the range of 0 to the larger row number of X and Y minus 1.

Examples

Compatibility

See also
alignsignals (users), *xcorr* (users)

findstr

Find a string within another, longer string

Syntax:
k = findstr(str1,str2)

Definition:
k = findstr(str1,str2) searches the longer of the two input strings for any occurrences of the shorter string, returning the starting index of each such occurrence in the double array, k. If no occurrences are found, then findstr returns the empty array, [].

The search performed by findstr is case sensitive. Any leading and trailing blanks in either input string are explicitly included in the comparison

Examples:

Formula	Result
s = 'Find the starting indices of the shorter string.';	
findstr(s,'the')	6 30
findstr('the',s)	6 30

Compatibility:
String, array

firls

Multiband least square FIR filter design

Syntax

H = firls(N,F,M)

H = firls(N,F,M,W)

Definition

1. H=firls(N,F,M,W) calculates an odd length-N+1 FIR filter which is a weighted least squares approximataion to an desired ideal response given by the band edges in the even length vector F with constant values given in the even length vectorm in each band. Each band may be weighted by values given in the vector W.
2. Please refer to:
   1) I. Selesnick, "Linear-Phase FIR Filter Design by Least Squares," http://cnx.org/content/m10577

   2) "multiband least squares filter design" by

C.Sidney Burrus in IEEE Transactions on Signal Processing
Vol.43,NO.2 1995

Examples

Compatibility

See also

firrcos

Raised cosine FIR Filter design

Syntax

H = FIRRCOS(N,Fc,DF)

H = FIRRCOS(N,Fc,DF,Fs)

H = FIRRCOS(N,Fc,R,Fs,R_OPTION)

H = FIRRCOS(N,Fc,R,Fs,R_OPTION,DESIGNTYPE)

H = FIRRCOS(N,Fc,R,Fs,R_OPTION,DESIGNTYPE,DELAY)

H = FIRRCOS(N,Fc,R,Fs,R_OPTION,DESIGNTYPE,DELAY,WINDOW)

Definition

1. Parameter definition

   N: an EVEN integer, the returned filter shall have N+1 taps.

   Fc: Cutoff frequency in Hz. When designing pulse shaping filter, Fc refers to the original symbol rate.

   DF: Transition bandwidth in Hz. When designing pulse shaping filter, DF refers to RolloffFactor*2*Fc.

   R: Rolling off factor whose relationship with DF is DF=R*Fc*2.

   Fs: The filter's operating rate in Hz.

   R_OPTION: 'rolloff' or 'bandwidth'.

   DESIGNTYPE: 'normal' or 'sqrt'.

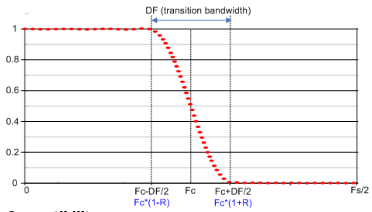
   DELAY: an integer in the range of [0,N+1]. Normally it's set to N/2.

   WINDOW: N+1 long column vector.
2. B = FIRRCOS(N,Fc,R,Fs,R_OPTION,DESIGNTYPE,DELAY,WINDOW) returns an order N low pass linear phase FIR filter with a raised cosine transition band or rolling factor R. The filter has cutoff frequency Fc, sampling frequency Fs and transition bandwidth DF (all in Hz). The order of the filter, N, must be even. R must be in the range [0,1]. If R_OPTION is set to 'rolloff', then R represents the rolling facor, otherwise it represents transition bandwidth. If DESIGNTYPE is set to 'sqrt', then it will be the square raised cosine filter, otherwise, it's the normal raised cosine filter design. DELAY must be an integer in the range [0, N+1], if not specified Delay will be N/2. WINDOW must be a N+1 long column vector. The time domain raised cosine filter can be implemented in frequency domain as an alternative way.
3. H = FIRRCOS(N,Fc,DF) performs as H = FIRRCOS(N,Fc,DF,2,'bandwidth','normal',N/2,ones(N+1,1)).
4. H = FIRRCOS(N,Fc,DF,Fs) performs as H = FIRRCOS(N,Fc,DF,Fs,'bandwidth','normal',N/2,ones(N+1,1)).
5. H = FIRRCOS(N,Fc,R,Fs,R_OPTION) performs as H = FIRRCOS(N,Fc,DF,Fs,R_OPTION,'normal',N/2,ones(N+1,1)). If R_OPTION='bandwidth', R refers to the transition bandwidth. If R_OPTION='rolloff', R refers to the rolling off factor within [0,1].
6. H = FIRRCOS(N,Fc,R,Fs,R_OPTION,DESIGNTYPE) performs as H = FIRRCOS(N,Fc,DF,Fs,R_OPTION,DESIGNTYPE,N/2,ones(N+1,1)). If DESIGNTYPE='normal', the filter will be raised cosine type. If DESIGNTYPE='sqrt', then the filter will be square root raised cosine type. In communication systems, in general, two square root raised cosine filters are used , one in the transmitter side and the other in the receiver side. The convolution of two square root raised cosine filter equals to one normal raised cosine filter plus a delay.

Examples

```
% example for communication baseband pulse shaping with upsampling ratio of 2
x = randint(100,1,[0,1])*2-1;
Ratio = 2; % upsampling ratio
N = 20*Ratio; % filter taps size, 20 is the number of input symbols used for interpolation
filtering
Fc = 3.84e6; % cutoff frequency, i.e. input symbol rate for communication modulation pulse
shaping.
DF = 0.22*2*Fc; % WCDMA uses 0.22 as rolling off factor
Fs = Fc*2*Ratio; % output sampling rate, *2 means Fs>=2*Fc according to Nyquist theory,
                % *Ratio refers to oversampling
R_OPTION = 'bandwidth';
DESIGNTYPE = 'sqrt';
H = Ratio*firrcos(N,Fc,DF,Fs,R_OPTION,DESIGNTYPE);
 % symmetric raised cosine filter with rectangular window,
 % Ratio* is to make output power near that of input
x_up = upsample(x,Ratio);
y = filter(H,1,x_up);
```

The frequency response of the designed filter (DESIGNTYPE='normal') is shown below



Compatibility

See also
firls (users)

fix

Syntax
y = fix(x)

Definition
fix rounds the argument toward zero, producing integer. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
fix(2.2)	2
fix(2.2 + 3.3j)	2 + 3j
fix(-2.3 - 3.9j)	-2 - 3j

Compatibility
Numeric scalars, Vectors, Arrays

See Also
floor (users)

floor

Syntax
y = floor(x)

Definition
floor returns the largest integer less than or equal to the argument. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
floor(10)	10
floor(1.5 + 6.2j))	1
floor([-0.5, 0.5])	[-1 , 0]

Compatibility
Numeric scalars, Vectors, Arrays

See Also
ceil (users)

fopen

Opens a file to read, write or append.

Syntax

fileP = fopen(filename)

fileP = fopen(fileName, operationFormat)

fileP, mess = fopen(filename, operationFormat)

Definition

This function performs file access and returns a handle *fileP* to the beginning of a file whose name is *filename* enclosed in single quotes. The file name can be an absolute path or a relative path. An extension for the file name is optional. The operationFormat is specified in *operationFormat*. Supported operations are:

'r'	Open for reading.
'a'	Open or create a file for writing. Append data the end of the file if content exists.
'w'	Open or create a file for writing. Truncate the file if content exists.
'r+'	Open for reading and writing.
'a+'	Open or create a file for reading and writing. Append data the end of the file if content exists.
'w+'	Open or create a file for reading and writing. Truncate the file if content exists.

If the fopen fails, *fileP* is -1 in contrast to a positive value if the operation was successful.

If two outputs are expected, the first one will be the handle *fileP* and the second one will be an appropriate message indicating whether the file was successfully opened or not.

Note that for binary files, the functions *fread* (users) and *fwrite* (users) should be used for file access.

Example

fileP = fopen('C:\TEMP\test.txt') will open the existing *test.txt* file for reading.

fileP = fopen('C:\TEMP\test.txt', 'w') will create the *test.txt* file and open it for writing.

See Also

fclose (users)
fgets (users)
fread (users)
fprintf (users)
fscanf (users)
fwrite (users)
tcpip (users)

fprintf

Syntax
count = fprintf(fid, format, A, ...)

Definition
Formats data from a matrix *A* or set of matrices ... and writes results to a file *fid*. *count* is the number of elements that were written to the file.

Compatibility
The first argument is a file handle which is returned from a call to fopen, followed by a format string and then by one or more matrix arguments.

The format string is of the form (only the leading % and *conversionChar* are required):
%{*Flags*}{*FieldWidth*}{.*Precision*}*ConversionChar*

Flags are used to control the alignment and padding of the output. Valid flags are:

Character	Description	Example
Minus sign (-)	Left-justify the output in its field	%-6.4d
Plus sign (+)	Always print a sign (+ or -) character; by default sign is printed only for negative numbers	%+6.4d
Space character	If no sign character is going to be printed insert a space before the value	% 6.4d
Zero (0)	Left-pads with zeros rather than spaces	%06.4d

For negative numbers, the '+' (plus) and ' ' (space) flags have no effect (they are ignored).
For positive numbers they determine whether a '+' or ' ' character will be printed before the value. If both flags are specified then the ' ' one is ignored.

FieldWidth specifies the minimum number of characters (including digits, the '.' character, sign characters, spaces, etc.) that will be printed for the field. If left empty, then it defaults to the minumum number of characters that is needed to print the result (no truncation or information loss will occur).

Precision specifies

- the total number of digits to be printed (for integer numbers)
- the number of digits to be printed after the decimal point (for floating point numbers using %f, %e, or %E)
- the number of significant digits to be printed (for floating point numbers using %g or %G)

ConversionChar must be one of the following:

Character	Description
c	Character sequence
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Use the shorter of %e or %f; do not print trailing zeros
G	Use the shorter of %E or %f; do not print trailing zeros
o	Unsigned octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)

When a numeric conversion character is used with a complex number, only the real part of the complex number is used.
When an unsigned integral conversion character (u, o, x, X) is used with a negative number, the conversion character is changed to e.

Differences with MATLAB

1. When the scientific notation is used, the exponent is printed using two digits (e.g. 1.2345e+02) unless three digits are needed to represent it (e.g. 1.2345e+105). On Windows platforms, MATLAB will use three digits to represent the exponent in scientific notation. This may result in one extra 0 at the beginning or one extra space at the beginning or end (if a *FieldWidth* is explicitly specified).
2. When floating point values are printed using an integral conversion character, they are first truncated to integers and then printed.

Value	Format	MATLAB	SystemVue
5.1235e-6	%d	5.123500e-006	0
81.2915	%x	8.129150e+001	51
81.2915	%d	8.129150e+001	81

3. When values need to be rounded due to a finite number of precision digits specified values are rounded down. For example, printing 1.2345e4 using %.3e results in 1.234e+04 in SystemVue and 1.235e+004 in MATLAB.
4. There is no support for the identifier operator ($).
5. There is no support for the '#' flag.
6. There is no support for '*' used in the *FieldWidth* or *Precision*.
7. There is no support for sub-type specification (e.g l, h, b, t)

See Also
fclose (users)
fgets (users)
fopen (users)
fread (users)
fscanf (users)
fwrite (users)
tcpip (users)

fread

Reads binary data from a file.
fread supports both TCP/IP and FILE I/O connections.

TCP/IP

Syntax
cIn = fread(cStream, iValues, cConvert)

Definition
Read some amount of binary data from the stream.

- cStream is a stream class object.
- iValues is the number of values to read.
- cConvert is a string array defining how to read. It can be 'type' to read as this type. It can be '*type' to have both input and output by this type. It can be 'type1=>type2' to have input data interpreted as type1 and output data in type2. By default, input format is byte and output format is double.

Examples:

Formula	Result
dOut = fread(t, 12, 'double')	read 12 doubles from the input and save them as doubles (the default). This will consume 96 bytes of input data.
iOut = fread(t, 100, '*int')	read 100 integers from the input and save them as integers. This will consume 400 bytes of input data.
cOut = fread(t, 22, 'uchar=>ushort')	read 22 ascii characters as input and save them as a character array \

FILE I/O

Description

Formula	Result
fread(fileP)	reads the contents of the file pointed to by the handle *fileP* (obtained from fopen.) The file is read from beginning to end and *fileP* is finally positioned at the end of the file.
mat=fread(fileP)	does exactly the above and returns a matrix *mat* with the contents of the file.

Compatibility
Scalars, Vectors, Arrays. Real and Complex and Character.

See Also:
fclose (users)
fgets (users)
fopen (users)
fprintf (users)
fscanf (users)
fwrite (users)
tcpip (users)
(2 bytes per character for unicode).
This will consume 22 bytes of input data.|

fscanf

Syntax
A = fscanf(fileP, format)
A = fscanf(fileP, format, size)

Definition
This function reads data from a file represented by a file handle *fileP* and converts it to a string using *format*. The result is returned in a matrix *A*.

An optional argument can be passed *size*, to specify the amount of data in the resulting matrix.

Compatibility
fileP - file pointer to an open file ready for reading
format - string description of format in which to access contents of file, e.g. '%f' for floating-point

size - positive integer specifying number of parts to be read in *readFormat*
size can be in the form:

n	read at most n elements from the file
inf	read to the end of the file
[m,n]	read at most m*n elements. Fill at most m rows in *A*

The format string consists of conversion patterns and characters to skip over. A conversion patter start with the % character and at a minimum a *conversion character*. Characters outside a conversion pattern must match in the input but will be skipped in the output.

digit	Maximum field width
*	Skip over the match value for this format. The value much match but will be ignored and not added to *A*

Valid *conversion characters* are:

c	Character sequence
d or i	Signed decimal integer
e	Scientific notation (mantise/exponent) using e character
E	Scientific notation (mantise/exponent) using E character
f	Decimal floating point
g	Use the shorter of %e or %f
G	Use the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)

For the floating point conversion characters (e, E, f, g and G), fscanf will also accept Inf, -Inf, NaN and -NaN (not case sensitive) as unputs.

See Also
fclose (users)
fgets (users)
fopen (users)
fprintf (users)
fread (users)
fwrite (users)
tcpip (users)

fwrite

Writes binary data to a file.
fwrite supports both TCP/IP and FILE I/O operations.

TCP_IP

Syntax
iWritten = fwrite(cStream, Value)

iWritten = fwrite(cStream, Value, Mode)

iWritten = fwrite(cStream, Value, Precision, Mode)

Definition
Write some amount of binary data to the stream.

- cStream is a stream class object.
- Value is the data to write.
- Mode can be 'sync' or 'async', default is async.
- Precision is a char array defining the output data type. Default is byte.

Examples:

Formula	Result
iOut = fwrite(t, 12)	Write the value 12 to the stream as a single byte.
iOut = fwrite(t, [1, 2], 'int', 'async')	Write the vector [1 2] to the stream as two integers.
iOut = fwrite(t, 22, 'sync')	Write the value 22 synchronously to the stream as a single byte.

FILE I/O

Description

Formula	Result
fwrite(fileP, mat)	will write the contents of matrix *mat* to the file pointed to by the handle *fileP* (obtained from fopen.) Data is written to the file in column order.
counter=fwrite(fileP, mat)	will do exactly the above. In addition it will return a counter with the number of elements successfully written to the file.

Note: Until the file is closed using the *fclose* (users) function, the contents of that fill cannot be viewed.

Compatibility
Scalars, Vectors, Arrays. Real and Complex and Character.

See Also
fclose (users)
fgets (users)
fopen (users)
fprintf (users)
fread (users)
fscanf (users)
tcpip (users)

gaussfir

Gaussian FIR Pulse-Shaping Filter Design

Syntax

H=gaussfir(BT)

H=gaussfir(BT,NT)

H=gaussfir(BT,NT,OF)

Definition

1. H=gaussfir(BT) designs a low pass FIR gaussian pulse-shaping filter. BT is the 3-dB bandwidth-symbol time product where B is the one-sided bandwidth in Hertz and T is in seconds.
2. H=gaussfir(BT,NT) NT is the number of symbol periods between the start of the filter impulse response and its peak. If NT is not specified, NT = 3 is used.
3. H=gaussfir(BT,NT,OF) OF is the oversampling factor, that is, the number of samples per symbol. If OF is not specified, OF = 2 is used.
4. The length of the impulse response of the filter is given by 2*OF*NT+1. Also, the coefficients H are normalized so that the nominal passband gain is always equal to one.

Examples

Compatibility

See also

gausswin

Syntax
c = gausswin(L)
c = gausswin(L,alpha)

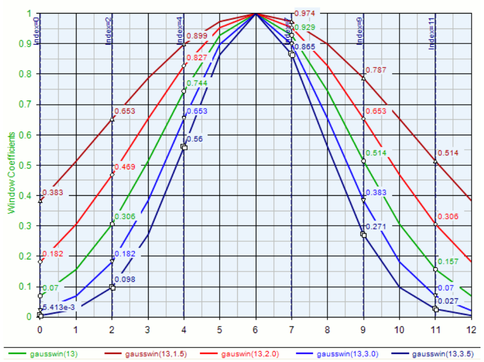
Definition
This function returns, in the column vector c, a Gaussian window with L-points and a window width parameter *alpha*. The default value of *alpha* is 2.5. The width of the window is inversely related to the value of *alpha* as shown in the graph below.

```
_gausswin_at_n_of_L_with_alpha_ = exp( -0.5 * (2 * alpha * n / N ) ^ 2 )
where -N/2 <= n <= N/2, L = N+1
```

When L is odd valued the apex of 1 is reached by the central sample. When L is even the two samples flanking the unsampled apex have a value of less than 1.

> **Note**
> gausswin(2), a redundant usage of this function returns [0.458 0.458], whereas gausswin(1) returns [1].

Examples:
In the following graph, 13-point Gaussian windows are overlaid for *alpha* in the range [1.5,3.5]. Vector values at each sample point are shown. The default behavior of *alpha* =2.5 is shown in green.
Note that the values at the end points of the vector are not forced to zero but rather determined by the value of *alpha*.



Compatibility
scalar

See Also:
bartlett (users)
blackman (users)
hamming (users)
hann (users)
rectwin (users)

getindep

Syntax
y = getindep(x)

Definition
Returns a string with the name(s) of the independent variable(s). x is the variable to check.

Examples:

Formula	Result
n=getindep(S)	if S is a linear analysis result this will usually return "Linear_Data\Eqns\VarBlock\F" (the longname of F)
n=getindep(VPORT)	in a HARBEC analysis this will return "HbData\Eqns\VarBlock\Freq" - the Frequency vector

Compatibility
Swept vectors, arrays

See Also
setindep

getindepvalue

Syntax
y = (x)

Definition
text

Examples:

Formula	Result

Compatibility
text

See Also
text

getmatlabvariables

Syntax
getmatlabvariables('var1', 'var2')
getmatlabvariables var1 var2

Definition
This function gets a list of MATLAB variables to SystemVue.

Examples:

Formula	Result
getvariable('var1', 'var2')	define SystemVue variables(var1, var2) and set the values from MATLAB variables

Compatibility
variable is string.

See Also
MATLAB Integration, *setmatlabvariables* (users)

getunits

Syntax
y = getunits(x)

Definition
Returns an integer corresponding to the units of a variable x. This integer may be used by setunits.

Examples:

Formula	Result
z = 1 setunits("z" , "V") y = getunits(z)	y = 9001
z = 1 setunits("z" , "mil") y = getunits(z)	y = 6002
z = 1 setunits("z" , "H") y = getunits(z)	y = 4003

Compatibility
Numeric scalars, vectors, arrays

See Also
setunits

getvariable

Syntax
y = getvariable(Dataset, Variable)
[y, yindep] = getvariable(Dataset, Variable)

Definition
This function gets a variable value (and, optionally, the value of its independent variable) from a dataset. The Dataset and Variable arguments must be strings. If an independent value is requested but the referenced variable doesn't have one, a warning is issued and yindep is set to a blank value.

Examples:

Formula	Result
OutVar = getvariable('OutData', 'OutVar')	set the variable OutVar from the dataset variable OutData.OutVar
myVar = getvariable('Out', 'Var')	set the variable myVar from the dataset variable Out.Var
[myVar, myIndep] = getvariable('Out', 'Var')	set the variable myVar from the dataset variable out.Var and set myIndep to Out.Var's independent value

Compatibility
Dataset and Variable are strings.

See Also
setvariable (users)

grpdelay

Group delay of IIR filter

Syntax

[H_R, W_R] = grpdelay(B, A, N, REGION)

[H_R, W_R] = grpdelay(B, A, F)

Definition

1. [H_R, W_R] = grpdelay(B, A, N, REGION) turns the group delay specified by the numerator B, the denominator A, number of frequcenies N. REGION has two options: half or whole, which determines the frequency sample point to be even distributed in pi or 2*pi.
2. [H_R, W_R] = grpdelay(B, A, F) turns the group delay at the frequency vector F.

Examples

Compatibility

See also
fft (users), *hilbert* (users)

hamming

Syntax
c = hamming(L)

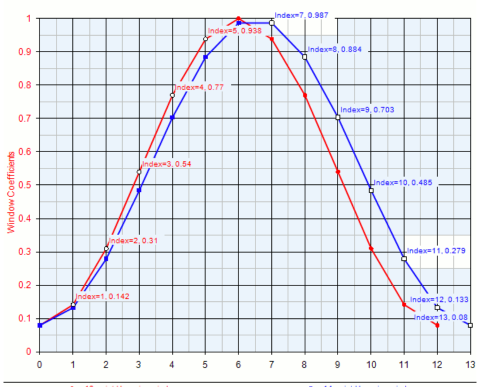
Definition
This function returns a Hamming window with L points into a column vector, c.

```
_hamming_value_at_n_of_L_symmetric_ = 0.54 - 0.46 * cos( 2*pi*n/N )
where 0 <= n <= N
```

Note that the end points of the vector is not always 0. When N is odd, the apex of 1 is explicitly an part of the window function. When N is even, the apex is not explicitly sampled but rather the two sample points which flank the apex are represented in the returned vector.

> **Note**
> hamming(2) a redundant usage of this function returns [0.08 0.08] whereas hamming(1) returns [1].

Examples:



Compatibility
scalar

See Also:
bartlett (users)
blackman (users)
gausswin (users)
hann (users)
rectwin (users)

hann

Syntax
c = hann(L)

Definition

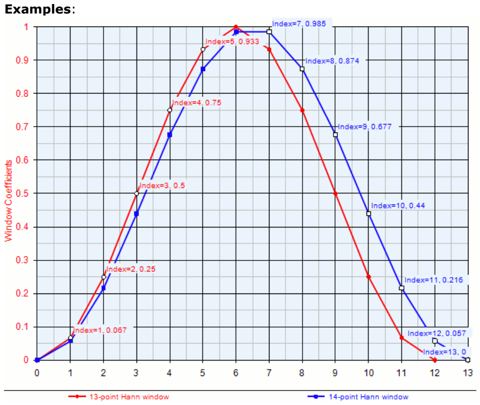
This function returns a hann window with L points into a column vector, c.

```
_hann_value_at_n_of_L_symmetric_ = 0.5 * ( 1 - cos( 2*pi*n/N) )
where 0 <= n <= N
```

Note that the end points of the vector are always 0. When N is odd, the apex of 1 is explicitly an part of the window function. When N is even, the apex is not explicitly sampled but rather the two sample points which flank the apex are represented in the returned vector.

> **Note**
> hann(2) a redundant usage of this function returns [0 0] whereas hann(1) returns [1].

Examples:



Compatibility
scalar

See Also:
bartlett (users)
blackman (users)
gausswin (users)
hamming (users)
rectwin (users)

hex2dec

Convert a hexadecimal string to decimal number

Syntax:
hex2dec('hex_value')

Definition:
The hex2dec function converts a hexadecimal number string to its decimal equivalent.

Examples:

Formula	Result
d = hex2dec('3ff')	d stores the conversion of hex no '3ff' in decimal i.e. d = 1023
d = hex2dec(S) where S is a character array S[0]= 0FF, s[1]=2DE & s[2]=123	255, 734, 291

Compatibility:
This function is Excel compatible.

See Also:
dec2hex (users)

hilbert

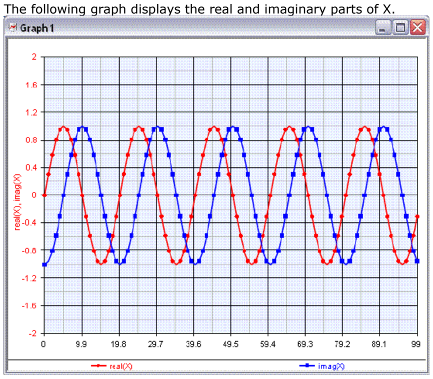
Syntax:
V = hilbert(X)
V = hilbert(X,n)
V = hilbert(X,[],c)
V = hilbert(X,n,c)

Definition:
Computes the *analytic signal* from a real data vector using the Hilbert Transform, where the Discrete Fourier Transform is used to calculate the Hilbert Transform. In the resulting complex vector the original real vector values are stored in the real part, and the imaginary part is the Hilbert Transform of the real vector.
hilbert(X) calculates the *analytic signal* of vector X. If X is a matrix, the *analytic signal* is computed for each column.
hilbert(X,n) returns the n-point *analytic signal*. X is extended by adding zeros if n>length of X, X is truncated if n<length of X.
hilbert(X, [], c) calculates the *analytic signal* on the dimension c.

Examples:
The following example creates a simple sinusoid at 400Hz then generates the analytic signal from that waveform. The resulting complex vector will contain the original sine wave as the real part, and a cosine wave (the Hilbert transform) as the imaginary part.

```
fs = 8000                        % 8000 Hz sampling rate
T = 1/fs                         % sample time
L = 100                          % length of signal
t = (0:(L-1))*T                  % time vector
x = sin(2*pi*400*t)              % sine wave at 400 Hz
X = hilbert(x)                   % analytic signal calculation
```

The following graph displays the real and imaginary parts of X.



Compatibility:
Vectors, Arrays, Dataset

See Also:

histc

Syntax
y = histc(x,e)
y = histc(x,e,dim)
[y,bin] = histc(x,e)

Definition
This function provides a count of the number of parts of a numeric real-valued vector or array *x* that fall into each histogram bin where the histogram itself is defined by the bin boundaries defined in the vector *e*. By definition *y* is a vector of integers. If *x* is a multi-dimensional array then the dimension of binning may be included as an optional third argument *dim*. If *dim* is omitted, the innermost non-singleton dimension is chosen as the dimension to operate along.

If the function is called with an optional *bin* output variable, then the actual binning matrix is returned in addition to the bin count vector *y*.

Examples:

```
% Define a large vector of normally-distributed random variables, with mean = 0
x = randn( 1, 1e5 );
% Detect the number that is farthest from 0
elim = max( abs( x ) );
% Create binning vector with bin span being one
e = [-elim-1:elim+1]
% Invoke histogram count
y = histc( x, e );
% y = [0,4,151,2400,14546,34846,33406,12648,1881,117,1,0]
% Note that the normalized distribution of values is captured in the vector y.
%
% Define a large 2-D array of bi-normally distributed random variables with mean = 0,0
x = randn( 3, 1e3 );
% Detect the number that is farthest from 0 along any radius
elim = max( abs( x );
% Create binning vector with bin span being one
e = [-elim-1:elim+1]
% Invoke histogram count row-wise (along dim=2)
y = histc( x, e, 2 );
% y = [0,0,2,1,5,25,65,109,174,182,172,140,78,35,7,4,0,0,1,0;
%      0,0,0,2,11,13,59,113,156,206,193,124,68,33,20,2,0,0,0,0;
%      0,0,0,4,9,27,59,118,162,189,180,127,81,31,8,3,2,0,0,,0];
%
% Note that the normalized distribution of values is captured in each row of y.
```

ifft

Syntax
ifft(data, len)

Definition
Inverse Discrete Fourier Transform (IDFT) of data. Computed with IFFT algorithm when possible. The parameter len is the IFFT length and is optional.

Examples:
The following example code is taken from the fft example:

```
fft_len = 1024                          % length of the FFT
fs = 8000                                     % 8000 Hz sampling rate
T = 1/fs                                      % sample time
L = 1000                                      % length of signal
t = (0:(L-1))*T                               % time vector
                                             % x will be the sum of two sinusoids:
                                             % one at 400 Hz and one at 1500 Hz
x = 0.5*cos(2*pi*400*t) + cos(2*pi*1500*t)
X = fft(x, fft_len)                           % spectrum of x
X = X(1:(fft_len/2))                          % we only care about single side\-band  (the rest is
redundant)
f = fs/2 * (0:(2/fft_len):1)
```

If the following lines of code are now added:
y = ifft(X, fft_len)
then y and x would be identical.

Compatibility
Dataset

See Also
fft (users)

imag

Syntax
y = imag(x)

Definition
imag returns the imaginary part of a complex number. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
imag(2 - 5j)	-5
imag([10 + 1j , 12])	[1 , 0]
imag([20 + 3j; 1 + 2j])	[3 ; 2]

Compatibility
Numeric scalars, Vectors, Arrays

impz

Impulse response of IIR digital filter

Syntax

[H,T] = impz(B,A,N)

[H,T] = impz(B,A,N,Fs)

Definition

1. [H,T] = impz(B,A,N) generate N samples of the impulse response of the filter with numerator coefficients B and denominator coefficients A. If N is not spcified, it will be choosen that the signal has a chance to die down to 10e-6 of the original, or to not explode beyond 10e6 of the original. Fs is the sampling frequency; the default value is 1.

Examples

Compatibility

See also

inf

Syntax
DA = Inf(n, dist)
DA = Inf(m, n, dist)
DA = Inf(..., classname, dist)

Definition
this function creates an n by n, or m by n, array of class double.
The classname parameter is for specifying the underlying class, which can be either 'double', the default, or 'single'.

Examples:

Formula	Result
x = inf	1.#IOe

Compatibility
Numeric

interp

Resample input at a higher rate with lowpass filter

Syntax

Y = interp(X,R)

Y = interp(X,R,L)

Y = interp(X,R,L,Alpha)

Y = interp(X,R,L,Alpha,SNR)

[Y,B] = interp(...)

Definition

1. Y = interp(X,R) resamples the signal in vector X at R times the original sample rate. The resampled vector Y is R times the length of X. Filter transition is compensated by image the input signal at the beginning and the end of filtering.
2. The symmetric lowpass filter B is obtained with minimum mean square error (MSE) rule, it allows the original signal pass through unchanged and minimizes the mean square error between the interpolated signal and the expected signal.
3. Y = interp(X,R,L,ALPHA,SNR) is used for specific filter length, cutoff frequency and signal noise power ratio. 2*L is the number of original samples used to compute each new sample. The filter length is 2*L*R+1. Alpha is bandwidth of the input signal which should satisfy 0 < ALPHA <1.0, where 1.0 correspons to half the sample rate. SNR is the power ratio of useful signal and AWGN noise. By default, L, ALPHA and SNR is 4, 0.5 and 150(dB) respectively. For some large L, try a lower SNR to get a reliable filter B.
4. [Y,B] = interp(X,R,L,ALPHA,SNR) returns the output and coefficients of filter B.

Examples

Compatibility

See also
resample (users), *upfirdn* (users)

interp1

Syntax
y2 = interp1(x1,Y1,x2)
y2 = interp1(x1,Y1,x2,method)
y2 = interp1(x1,Y1,x2,method,'extrap')
pp = interp1(x,Y,method,'pp')

Definition
y2 = interp1(x1,Y1,x2) interpolates to find y2, the values of the underlying function Y1 at the points in the vector x1.
method:
'nearest' Nearest neighbor interpolation
'linear' Linear interpolation (default)
'spline' Cubic spline interpolation
'pchip' Piecewise cubic Hermite interpolation
'cubic' (Same as 'pchip')

yi = interp1(x,Y,xi,method,'extrap') uses the specified method to perform extrapolation for out of range values.

Examples:

Formula	Result
x1=[1 2 4 5] y1=[34 56 67 77] y2=interp1(x1,y1,3)	y2 ? ans = ? 61.5
x1=[1 2 4 5] y1=[34 56 67 77] y2=interp1(x1,y1,-1,'linear','extrap')	y2 ? ans = ? -10

Compatibility
Numeric scalars, Vectors

See Also
spline (users)

ischar

Syntax
y = char(x)

Definition
This function determines whether the given parameter *x*, is a character or array of characters. If so, it returns true, logical 1, and if not it returns false, logical 0.

Examples:

Formula	Result	Comment
ischar(2)	0	scalar is not a character
ischar('2')	1	character
ischar(1:10)	0	numeric vector is not an array of characters
ischar('hello')	1	vector of characters
ischar(['hello','table'])	1	array of characters
ischar({'hello','table'})	0	cell is not an array

Compatibility
Numeric and string valued variables.

See Also:
isempty (users)
isfloat (users)
isinteger (users)
islogical (users)
isreal (users)
isscalar (users)
isstr (users)

isempty

Syntax
y = isempty(x)

Definition
This function returns true if *x* is an empty array and false otherwise. An empty array has at least one dimension of size zero, for example, 0-x-0 or 0-x-5. This function does not operate on strings or cells. So supplying an empty string to the function does not get a logical true.

Examples:

Formula	Result
isempty(rand(2,2))	0
b(:,:) = []; a = isempty(b)	a = 1;

Compatibility
Numeric scalars, vectors, arrays.

isequal

Syntax
out = isequal(a, b[, ...])

Definition
isequal returns true if the input arrays have the same contents, and false otherwise. Nonempty arrays must be of the same data type and size to be compared.

Examples:

Formula	Result
a = [1,2;3,4] b = [1,2;3,4] out = isequal(a,b)	out=1;

Compatibility
Arrays and scalars.

isfinite

Syntax
b = isfinite(Array)

Definition
isfinite returns an array the same size as Array containing true where the parts of Array are finite and false where they are infinite or NaN. For a complex number z, isfinite(z) returns true if both the real and imaginary parts of z are finite, and false if either the real or the imaginary part is infinite or NaN.

Compatibility
Numeric arrays

See Also
isinf (users)

isfloat

Syntax
y = isfloat(x)

Definition
This function determines whether the given parameter *x*, is a floating point number. If so, it returns true, logical 1, and if not, it returns false, logical 0. When *x* is a character or a string, this function returns 0 because the argument is not an explicit numeric value but *isreal* (users) returns 1 because the argument is implicitly real valued because ASCII characters are involved in scalar or vector format.

Examples:

Formula	Result	Comment
isfloat(23)	1	scalar is a 1-part vector
isfloat(1:0.5:10)	1	row-vector of floating point numbers
isfloat([2+3i;4])	1	column-vector of real and complex numbers
isfloat('h')	0	ASCII character is not a floating point numeric value
isfloat('hello')	0	string is a vector of ASCII characters, not numeric values

Compatibility
Numeric and string valued variables.

See Also:
ischar (users)
isempty (users)
isinteger (users)
islogical (users)
isreal (users)
isscalar (users)
isstr (users)

isinf

Syntax
out = isinf(Array)

Definition
isinf returns an array the same size as Array containing true where the parts of Array are +Inf or -Inf and false where they are finite. For a complex number z, isinf(z) returns true if either the real or imaginary part of z is infinite, and false if both the real and imaginary parts are finite or NaN. For any real a, exactly one of the three quantities isfinite(a), isinf(a), and isnan(a) is true.

isinteger

Syntax
y = isinteger(x)

Definition
This function determines whether the given parameter *x*, is an integer. If so, it returns true, logical 1, and if not it returns false, logical 0. When applied to a multi-part array, all parts must be integers for the function to evaluate to a true.

Examples:

Formula	Result	Comment
isinteger(-23)	1	is an integer
isinteger(1:10)	1	10-part row-vector of integers
isinteger(1:0.5:2)	0	contains some non-integers

Compatibility
Numeric and string valued variables.

See Also:
ischar (users)
isempty (users)
isfloat (users)
islogical (users)
isreal (users)
isscalar (users)
isstr (users)

islogical

Syntax
y = islogical(x)

Definition
This function determines whether the given expression *x*, is evaluates to a binary logical value. If so, it returns true, logical 1, and if not it returns false, logical 0.

Examples:

Formula	Result	Comment
islogical(1)	0	numeric scalar not a logical expression even though it is binary valued 1
islogical(2>3)	1	is a logical expression
islogical([3,4] < [5,6])	1	is a logical expression

Compatibility
Numeric and string valued variables.

See Also:
ischar (users)
isempty (users)
isfloat (users)
isinteger (users)
isreal (users)
isscalar (users)
isstr (users)

isnan

Syntax
out = isnan(Array)

Definition
isnan returns an array the same size as Array containing true where the parts of Array are NaN (not-a-number). For any real a, exactly one of the three quantities isfinite(a), isinf(a), and isnan(a) is true.

isreal

Syntax
y = isreal(x)

Definition
This function determines whether the given parameter *x*, is a real valued number or a vector or array containing only real numbers. If so, it returns true, logical 1, and if not, it returns false, logical 0.

Examples:

Formula	Result	Comment
isreal(23)	1	integer is real valued
isreal(1:0.5:10)	1	10-part real-valued row-vector
isreal([3;4+5i;6])	0	has one complex valued part
isreal('h')	1	character has an ASCII value
isreal('hello')	1	string is an array of ASCII values
isreal({'hello'})	0	cell is not a numeric part
isreal({1.4})	0	cell is not a numeric part

Compatibility
Numeric and string valued variables.

See Also:
ischar (users)
isempty (users)
isfloat (users)
isinteger (users)
islogical (users)
isscalar (users)
isstr (users)

isscalar

Syntax
y = isscalar(x)

Definition
This function determines whether the given parameter *x*, is a 1x1 part with an ASCII value i.e. a scalar. If so, then it returns true, logical 1, and if not then it returns false, logical 0.

Examples:

Formula	Result	Comment
isscalar(23)	1	is a scalar
isscalar(1:10)	0	10-part row-vector
isscalar('d')	1	is an ASCII character
isscalar('hello')	1	string is not a scalar
isscalar({1})	1	is a 1-part cell
isscalar({'This is a sentence'})	1	is also a 1-part cell
isscalar({'This','is','a','sentence','.'})	0	is a multi-part cell

Compatibility
Numeric and string valued variables.

See Also:
ischar (users)
isempty (users)
isfloat (users)
isinteger (users)
islogical (users)
isreal (users)
isstr (users)

isstr

Syntax
y = isstr(x)

Definition
This function determines whether the given parameter *x*, is a string. If so, then it returns true, logical 1, and if not then it returns false, logical 0.

Examples:

Formula	Result	Comment
isstr(23)	0	scalar is not a string
isstr('hello')	1	is a string
isstr({'This','is','a','sentence','.'})	0	array of cells is not a string
isstr({'This'})	0	even single string part in cell is not a string

Compatibility
Numeric and string valued variables.

See Also:
ischar (users)
isempty (users)
isfloat (users)
isinteger (users)
islogical (users)
isreal (users)
isscalar (users)

kaiser

Kaiser window

Syntax

W = kaiser(NL,Beta)

Definition

1. W = kaiser(NL,Beta) returns column vector W of length NL for kaiser window. Beta affects the side attenuation of the spectrum. If NL is 1, it returns 1.

Examples

Compatibility

See also
gausswin (users), *kaiserord* (users)

kaiserord

Parameters that specify a kaiser window

Syntax

[n,wn,beta,ftype] = kaiserord(f,a,dev)

[n,wn,beta,ftype] = kaiserord(f,a,dev,Fs)

Definition

1. KAISERORD returns n, wn, beta and ftype that specify a kaiser window. It estimates the minimum filter order n and beta that can meet the specifications.
2. [N,WN,BETA,FTYPE] = kaiserord(F,A,DEV) F is band edge vector. A is a vector specifying the amplitude on the bands. 1 means passband and 0 means stopband. The length of F should be twice of the length of A, minus 2. DEV is a vector with the same size of A. It specifies the maximum deviation of the passband ripple and the stopband attenuation. The default value of Sampling frequency is 2 Hz and F should fall in [0,1].
3. [N,WN,BETA,FTYPE] = kaiserord(F,A,DEV,Fs) Fs is sampling frequency and F should fall in [0,Fs/2].

Examples

Compatibility

See also
kaiser (users)

length

Syntax
y = length(x)

Definition
This function returns the longest dimension of the array *x*. When presented with a single string, it returns the character count. When presented with a list of strings it returns list length even if one of the words has a character count (inner dimension) greater than the word count of the string (outer dimension).

Examples:

Formula	Result	Comment
length(16)	1	scalar number
length([1 2 3])	3	3-length vector
length([1 2 3; 4 5 6])	3	2x3 matrix, number of colmns > number of rows
length('hello')	5	string length is 5
length({'This','string','is','a','test','string','.'})	7	word count is 7, all words have character count < 7
length({'This','string','is','a','verylongwordedtest','string','.'})	7	word count is 7, even though one word has character count > 7

Compatibility
Numeric and string scalars, vectors, arrays

See Also
size (users)

linspace

Syntax
y = linspace(u,v)
y = linspace(u,v,x)

Definition
This function creates vectors that have values that are linearly spaced, similar to the colon operator. However, unlike the colon operator, this function gives control on specifying the number of points. The points are generated between, and including, *u* and *v*. The number of points generated are determined by the parameter *x*. If not specified, this value defaults to 100.

Examples:

Formula	Result
y = linspace(1, 10, 10)	Y = [1,2,3,4,5,6,7,8,9,10]

Compatibility
u - Real valued scalar
v - Real valued scalar
x - Positive integer

See Also:
logspace (users)

log

Syntax
y = log(x)

Definition
This function returns the natural logarithm (base e) of the argument *x*. It operates on an part-by-part basis on arrays. Exceptions of "-1.#INF" (negative infinity) and "-1.#IND"(indefinable) are thrown for zero and negative arguments respectively, as is to be expected. For complex valued arguments, the returned y = a + bi is such that a is log(sqrt(real(x)^2+imag(x)^2)), i.e. the natural log of the magnitude and b is atan(imag(x)/real(x)), i.e. the argument assumed to be in natural log.

Examples:

Formula	Result
log(1)	0
log([10 , 1.5])	[2.3 , 0.4]
log([2.3 , 0.5 ; 3.7 , 0.8])	0.832909 -0.693147 1.30833 -0.223144

Compatibility
Real and complex-valued scalars, Vectors, Arrays

See Also:
log10 (users)
log2 (users)

log2

Syntax
y = log2(x)
[f, e] = log2(x)

Definition
When used with one output argument, this function returns the base-2 logarithm of the

argument. It operates on an part-by-part basis on arrays. Exceptions of "-1.#INF" (negative infinity) and "-1.#IND"(indefinable) are thrown for zero and negative arguments respectively, as is to be expected. For complex valued arguments, the returned y = a + bi is such that a is log2(sqrt(real(x)^2+imag(x)^2)), i.e. the base-2 log of the magnitude and b is atan(imag(x)/real(x))/log(2), i.e. the argument assumed to be in base-2 log.

When used with two output arguments, the mantissa and exponent of the floating point argument are returned into f and e respectively.

Definition
This function returns the natural logarithm (base e) of the argument *x*.
Examples:

Formula	Result
log2(2)	1
log2([4, 512])	[2, 9]

Compatibility
Real and complex-valued scalars, vectors, arrays

See Also
log (users)
log10 (users)

log10

Syntax
y = log10(x)

Definition
This function returns the 10-base logarithm of the argument *x*. It operates on an part-by-part basis on arrays. Exceptions of "-1.#INF" (negative infinity) and "-1.#IND"(indefinable) are thrown for zero and negative arguments respectively, as is to be expected. For complex valued arguments, the returned y = a + bi such that a is log10(sqrt(real(x)^2+imag(x)^2)), i.e. the log10 of the magnitude of the vector and b is atan(imag /imag)/log(10) i.e. the log10 of the argument, where log(10) is the natural logarithm of 10.

Examples:

Formula	Result
log10(1)	0
log10([10 , 1.5])	[1 , 0.176]
log10([2.3 , 0.5 ; 3.7 , 0.8])	[0.362 , -0.301 ; 0.568 , -0.097]
log10(3+2i)	0.556972 + 0.255366i

Compatibility
Real and complex valued scalars, vectors, arrays

See Also
log (users)
log2 (users)

logspace

Syntax
y = logspace(u,v)
y = logspace(u,v,x)
y = logspace(u,pi)

Definition
This function creates a real-valued vector that is spaced logarithmically. It is the logarithmic equivalent of linspace and the colon operator (:), and is useful for generating frequency vectors.

This function generates values that are spaced from 10^u to 10^v. It creates an x number of points, and if x is not specified, it defaults the value to 50.

If pi is specified instead of v then the values are spaced from 10^u to pi (approx. 3.14). This is useful for digital signal processing where frequencies go around the unit circle.

Examples:

Formula	Result
logspace(1,6,6)	[10, 100, 1000, 1e4, 1e5, 1e6]
logspace(-3,3,7)	[0.001, 0.01, 0.1, 1, 10, 100, 1000]
logspace(0,1,10)	[1, 1.29155, 1.6681, 2.15443, 2.78256, 3.59381, 4.64159, 5.99484, 7.74264, 10]
logspace(0,pi,5)	[1, 1.33134, 1.77245, 2.35973, 3.14159]

Compatibility
u - Real valued scalar
v - Real valued scalar
x - Positive integer

See Also
linspace (users)

lp2bp

Transform lowpass filter to bandpass filter

Syntax

[bt,at] = lp2bp(b,a,wo)

[at,bt,ct,dt] = lp2bp(a,b,c,d,wo)

Definition

1. LP2BP transforms analog lowpass filter with normalized cutoff frequency of 1 rad/s into bandpass filter with desired central frequency and passband.
2. [bt,at] = lp2bp(b,a,wo) is the transfer function form. B and A are polynomial coefficients. wo has two elements. wo(1) is low band edge and wo(2) is high band edge.
3. [at,bt,ct,dt] = lp2bp(a,b,c,d,wo) is the state-space form.

⚠ Output arguments should **NOT** be omitted

Examples

Compatibility

See also
bilinear (users), *lp2bs* (users), *lp2hp* (users), *lp2lp* (users)

lp2bs

Transform lowpass filter to bandstop filter

Syntax

[bt,at] = lp2bs(b,a,wo)

[at,bt,ct,dt] = lp2bs(a,b,c,d,wo)

Definition

1. LP2BS transforms analog lowpass filter with normalized cutoff frequency of 1 rad/s into bandstop filter with desired central frequency and stopband.
2. [bt,at] = lp2bs(b,a,wo) is the transfer function form. B and A are polynomial coefficients. wo has two elements. wo(1) is low band edge and wo(2) is high band

edge.

3. [at,bt,ct,dt] = lp2bs(a,b,c,d,wo) is the state-space form.

⚠ Output arguments should **NOT** be omitted

Examples

Compatibility

See also
bilinear (users), *lp2bp* (users), *lp2hp* (users), *lp2lp* (users)

lp2hp

Transform lowpass filter to highpass filter

Syntax

[bt,at] = lp2hp(b,a,wo)

[at,bt,ct,dt] = lp2hp(a,b,c,d,wo)

Definition

1. LP2HP transforms analog lowpass filter with normalized cutoff frequency of 1 rad/s into highpass filter with desired cutoff frequency.
2. [bt,at] = lp2hp(b,a,wo) is in transfer function form. B and A are polynomial coefficients. wo is the desired cutoff frequency.
3. [at,bt,ct,dt] = lp2hp(a,b,c,d,wo) is in state-space form.

⚠ Output arguments should **NOT** be omitted

Examples

Compatibility

See also
bilinear (users), *lp2bp* (users), *lp2bs* (users), *lp2lp* (users)

lp2lp

Transform lowpass filter with normalized frequency to desired frequency

Syntax

[bt,at] = lp2lp(b,a,wo)

[at,bt,ct,dt] = lp2lp(a,b,c,d,wo)

Definition

1. LP2LP transforms analog lowpass filter with normalized cutoff frequency of 1 rad/s into lowpass filter with desired cutoff frequency.
2. [bt,at] = lp2lp(b,a,wo) is in transfer function form. B and A are polynomial coefficients. wo is the desired cutoff frequency.
3. [at,bt,ct,dt] = lp2lp(a,b,c,d,wo) is in state-space form

⚠ Output arguments should **NOT** be omitted

Examples

Compatibility

See also
bilinear (users), *lp2bp* (users), *lp2bs* (users), *lp2hp* (users)

lu

Syntax
[L,U,P] = lu(A)

Definition
Let A be an m x n matrix and k=min(m,n).

[L,U,P] = lu(A) produces matrices L, U, and P such that L·U = P·A, where
L is a lower triangular (when m≤n) or lower trapezoidal (when m>n) m x k matrix with unit parts in the primary diagonal
U is an upper triangular (when m≥n) or upper trapezoidal (when m<n) k x m matrix
P is a permutation m x m matrix

Examples:

```
>> A=randn(3,3)+j*randn(3,3)
  A =
         0.723014 + 1.18447j      0.934672 + 0.460644j     0.441228 + 0.256457j
        -0.328791 - 0.851946j    -0.861837 + 2.87705j      0.955427 + 1.76944j
         0.179696 - 0.856819j     1.68603 - 0.932334j     -0.0821437 - 1.2154j
>> [L,U,P] = lu(A)
  L =
                1                        0                 0
        -0.647459 - 0.117631j            1                 0
        -0.459545 - 0.43222j     -0.150244 - 0.569137j     1
  U =
         0.723014 + 1.18447j      0.934672 + 0.460644j      0.441228 + 0.256457j
                0                -0.310862 + 3.28524j       1.21094 + 1.98739j
                0                        0                 -0.939387 + 0.080946j
  P =
        1     0     0
        0     1     0
        0     0     1
>> max( max( abs( L*U-P*A ) ) )
  ans =
        1.11022e-016
>> A = rand(6,3)
  A =
        0.60099      0.440156     0.864022
        0.127121     0.130142     0.348069
        0.946835     0.559306     0.632182
        0.766416     0.852558     0.260926
        0.857445     0.0636686    0.47777
        0.447486     0.372438     0.510765
>> [L,U,P] = lu(A)
  L =
        1             0            0
        0.905591      1            0
        0.634736     -0.192272     1
        0.809451     -0.902882    -0.756563
        0.134259     -0.124312     0.565567
        0.472613     -0.244116     0.424851
  U =
        0.946835     0.559306      0.632182
        0           -0.442834     -0.0947284
        0            0             0.444539
  P =
        0     0     1     0     0     0
        0     0     0     0     1     0
        1     0     0     0     0     0
        0     0     0     1     0     0
        0     1     0     0     0     0
        0     0     0     0     0     1
>> max( max( abs( L*U-P*A ) ) )
  ans =
        1.11022e-016
```

matdeintrlv

Reorder data by filling matrix by columns and emptying it by rows

Syntax

Y = matdeintrlv(X,Rows,Cols)

Definition

Y = matdeintrlv(X,Rows,Cols) rearranges the data in X by writting a temporary matrix column by column and then reading the matrix row by row to the output. Rows and Cols specifies the size of the temporary matrix. If X is a vector, it must have Rows*Cols elements. If X is a matrix, it must have Rows*Cols rows, each column is treated as an independent signal.

Examples

```
b = matdeintrlv([1 4 2 5 3 6; 7 10 8 11 9 12].',2,3)
b =
   1     7
   2     8
   3     9
   4    10
   5    11
   6    12
```

Compatibility

See also
matintrlv (users)

matintrlv

Reorder data by filling matrix by rows and emptying it by columns

Syntax

Y = matintrlv(X,Rows,Cols)

Definition

Y = matintrlv(X,Rows,Cols) rearranges the data in X by writting a temporary matrix row by row and then reading the matrix column by column to the output. Rows and Cols specifies the size of the temporary matrix. If X is a vector, it must have Rows*Cols elements. If X is a matrix, it must have Rows*Cols rows, each column is treated as an independent signal.

Examples

```
b = matintrlv([1 2 3 4 5 6; 7 8 9 10 11 12].',2,3)
b =
   1     7
   4    10
   2     8
   5    11
   3     9
   6    12
```

Compatibility

See also
matdeintrlv (users)

max

Syntax
y = max(x)
y = max(x,z)
y = max(x,dim)
[y, i] = max(...)

Definition
Returns the maximum part of a vector x. In the case of arrays, the function returns a row vector with the maximum part in each column. When dealing with multidimensional arrays, it treats the parts along the first non-singleton dimension, or the specified dim, as vectors and returns the maximum of each.

y = max(x,z) returns an array with the same dimensions as x and z containing the maximum parts from vectors x or z. The size of x and z have to be the same.

The dim argument is optional and specifies which dimension to operate along. For example, if dim is 1, this function operates on each column of the argument. If the argument is omitted, the first non-singleton dimension is chosen as the dimension to operate along.

[y, i] = max(...) also returns the indices of the maximum parts in a vector i. If more than one maximum of the same value exists, then only the first parts index is returned.

Examples:

Formula	Result
x = 10 y = max(x)	y = 10
x = [18 -20 23 54 4 71 -43] y = max(x)	y = 71
x = [27 86; complex(600 , -435), 34] y = max(x)	y = [600 - j435, 86]
x = [27 86; complex(1 , 1) , -34] y = max(x)	y = [27, 86]

Compatibility
Numeric Scalars, Vectors, Arrays

See Also
min (users)

mean

Syntax
y = mean(x)
y = mean(x,dim)

Definition
Returns the arithmetic mean of a vector x.

For matrices, this function operates separately on each column and returns a vector. For multi-dimensional arrays in general, this function operates on the dimension specified by dim, or the first non-singleton dimension if dim, is not specified.

Examples:

Formula	Result
y = mean([3 ; 4 ; 8 ; 9])	y = 6
y = mean([complex(1 , 2) ; complex(1 , 1) ; complex(2 , 1)])	y = 1.333 + j1.333
y = mean([1,2,3;4,5,6;7,8,9])	y = [4, 5, 6]

Compatibility
Numeric arrays

See Also
median (users)

median

Syntax

y = median(x)
y = median(x,iDim)

Definition
Returns the median of a vector x.

For matrices, this function operates separately on each column and returns a vector. For multi-dimensional arrays in general, this function operates on the dimension specified by iDim, or the first non-singleton dimension if iDim is not specified.

Examples:

Formula	Result
y = median([3 ; 4 ; 8 ; 9])	y = 6
y = median([complex(1 , 2) ; complex(1 , 1) ; complex(2 , 1)])	y = 2 + j1
y = median([1,2,3;4,5,6;7,8,9])	y = [4, 5, 6]

Compatibility
Numeric arrays

See Also
mean (users)
mode (users)

min

Syntax
y = min(x)
y = min(x,z)
y = min(x,dim)
[y,i] = min(...)

Definition
Returns the minimum part of a vector x. In the case of complex-valued arrays, the magnitude of each part is used.

For matrices, this function operates separately on each column and returns a vector. For multi-dimensional arrays in general, this function operates on the dimension specified by dim, or the first non-singleton dimension if dim is not specified.

The dim argument is optional and specifies which dimension to operate along. For example, if dim is 1, this function operates on each column of the argument. If the argument is omitted, the first non-singleton dimension is chosen as the dimension to operate along.

[y,i] = min(...) also returns the indices of the minimum valued parts in x. If there are more than one minimum parts of the same value, the index of the first one found is returned.

Examples:

Formula	Result
x = [10] y = min(x)	y = 10
x = [18 , -20 , 23 , 54 , 4 , 71 , -43] y = min(x)	y = -43
x = [27, 86 ; complex(600 , -435) , 34] y = min(x)	y = [27, 34]
x = [27, 86 ; complex(1 , 1) , -34] y = min(x)	y = [1+j1, -34]

Compatibility
Numeric Scalars, Vectors, Arrays

See Also
max (users)

mkdir

Make a new directory

Syntax:
mkdir('dname')
mkdir('pdir','dname')
status = mkdir('pdir','dname')
[status,mess,messid] = mkdir('pdir','dname')

Definition:
mkdir('dname') creates the directory dname in the current directory. The full path of the directory is displayed in warnings. A warning sign is displayed if the directory dname already exists.

mkdir('pdir','dname') creates the directory dname in the existing directory pdir. An error is displayed if the directory pdir is not an existing directory. A warning sign is displayed if the directory dname already exists.

status = mkdir('pdir,'dname') creates the directory dname and returns a status of logical 1 if the operation was successful. It returns an error message if the creation of dname failed.

[status, mess, messid] = mkdir('pdir','dname') creates the directory dname and returns a status of logical 1 if the operation was successful. If dname previously existed, mess and messid contain appropriate messages.

Examples:
To create a New Sub Directory called NewTest in the Current Directory, type
mkdir ('NewTest')

To create a New Sub Directory called NewTest in the parent directory 'C:\Documents and Settings', type
mkdir ('C:\Documents and Settings','NewTest')

To be notified if NewTest is already an existing directory, type
[status, mess, messid]= mkdir('C:\Documents and Settings','NewTest') which will display
status = 1
mess = Directory already exists.
messid = MATHLANG:MKDIR:DirectoryExists

See Also:
movefile
cd

mod

Syntax
m = mod(a,b)

Definition
This function applies the modulus operation on *a* by *b*.
It returns, m = a - (floor(a./b) .* b). If *b* is a scalar, then all parts of *a* are treated by its value. If *b* is nor

Examples:

Formula	Result
m = mod(13, 5)	m = 3
m = mod([1:5],3)	m = [1,2,0,1,2]

Compatibility
Real valued scalars, vectors, arrays

See Also
rem (users)

mode

Syntax
y = mode(x)
y = mode(x,iDim)
[y, n] = mode(x, ...)
[y,n,ca] = mode(x, ...)

Definition
Returns the mode of a vector x. If there are several values with equal maximum number of occurrences, the smallest value is returned.

For matrices, this function operates separately on each column and returns a vector. For multi-dimensional arrays in general, this function operates on the dimension specified by iDim, or the first non-singleton dimension if iDim is not specified.

[y, n] = mode(x, ...) also returns an array of same size as y which contains the number of occurrences of each part in y.

[y,n,ca] = mode(x, ...) also returns a cell array with the same size as y and n, and it contains, in each part, a sorted vector of the values that have the same frequency as each part in y.

Examples:

Formula	Result
y = mode ([8 ; 4 ; 8 ; 9])	y = 8
y = mode ([complex(1 , 2) ; complex(1 , 2) ; complex(2 , 1)])	y = 1 + j2
y = mode ([1,2,3;2,2,3;7,8,9])	y = [1, 2, 3]

Compatibility
Numeric arrays

See Also
mean (users)
median (users)

muxdeintrlv

Restore ordering of data with specified shift register group

Syntax

Y = muxdeintrlv(X,Delay)

Y = muxdeintrlv(X,Delay,InitState)

[Y,FinalState] = muxdeintrlv(X,Delay,...)

Definition

Y = muxdeintrlv(X,Delay), as a reverse process of function MUXINTRLV, resotres ordering of data in X with shift register group specified in the vector Delay which must be the same as that in MUXINTRLV. The length of Delay indicates the number of shift registers used.

Y = muxdeintrlv(X,Delay,InitState) initialize the shift registers specified in InitState instead of all zeros.

[Y,FinalState] = muxdeintrlv(X,Delay,...) returns the final state of shift registers in FinalState which may be used as initial state of the next process when dealing with consecutive data.

The total processing delay of the muxintrlv concatenated by muxdeintrlv is Td = max(Delay) * size(Delay)

muxdeintrlv is implemented by calling *muxintrlv* (users)

```
Y = muxintrlv(X, max(Delay)-Delay)
```

Examples

Compatibility

See also
muxintrlv (users)

muxintrlv

Reorder data with specified shift register group

Syntax

Y = muxintrlv(X,Delay)

Y = muxintrlv(X,Delay,InitState)

[Y,FinalState] = muxintrlv(X,Delay,...)

Definition

Y = muxintrlv(X,Delay) rearranges the data in X with shift register group specified in the vector Delay. The length of Delay indicates the number of shift registers used. Each value of Delay indicates the number that shift register can hold data. The input data is fed into the shift registers, from the first to the last, in sequence and periodicly. Assuming X={x1,x2,x3,...}, x1 is fed into branch 1, x2 is fed into branch 2, ... The output picks up data from output of each shift register, from the first to the last, in sequence and periodicly. Note that the data feeding to each shift register and data picking up from each shift register are synchronou. If X is a matrix, each column is treated as an independent signal. Delay is initialized with zeros before process begins.

Y = muxintrlv(X,Delay,InitState) initialize the shift registers specified in InitState instead of all zeros. InitState is a struct composed of variables InitState.value and InitState.index. InitState.value has the same number of columns as X, each stores the initial state of shift registers (from first to last). FinalState.index represents the index of the shift register into which the first symbol shall be fed.

```
Assuming Delay is [0,1,2,3], InitState.value=[1 2 3 4 5 6].', InitState.index=2, then we have
initial state:
 []          --FIFO 1
 [1]         --FIFO 2
 [2  3]      --FIFO 3
 [4  5  6],  --FIFO 4
and shall start processing from the second FIFO.
```

[Y,FinalState] = muxintrlv(X,Delay,...) returns the final state of shift registers in FinalState which may be used as initial state of the next process when dealing with consecutive data. FinalState is a struct composed of variables FinalState.value and FinalState.index. FinalState.value has the same number of columns as X, each stores the final state of shift registers (from first to last) after processing the corresponding column of X. FinalState.index represents the index of the shift register from which the next consecutive processing shall begin.

Examples

Compatibility

See also
muxdeintrlv (users), *convintrlv* (users)

NaN

Syntax
array = NaN(n, distdim)
array = NaN(m, n, distdim)
array = NaN(..., classname, distdim)

Definition
This function creates an n-by-n, or m-by-n as specified, distributed array, which is of class double by default. The distributed dimension dim and partition PAR are specified by distdim in the parameters, but if not then it automates to the second dimension and defaultPartition(n) is used.

array = NaN(..., classname, distdim) also allows you to specify the class of the array. These can either be 'double' or 'single.'

Examples:

Formula	Result

false

Syntax
false

Definition
false as a boolean value.

Examples:

Formula	Result
x=false	false

See Also
true (users)

noisebw

Equivalent two-sided noise bandwidth of lowpass filter

Syntax

NBW = noisebw(NUM, DEN)

NBW = noisebw(NUM, DEN, NumSamp)

NBW = noisebw(NUM, DEN, NumSamp, Fs)

Definition

NBW = noisebw(NUM, DEN, NumSamp, Fs) returns the two-sided equivalent noise bandwidth, in Hz, of a digital LOWPASS filter given in descending power of z by numerator vector NUM and denominator vector DEN. NumSamp specifies the number of impulse response samples used for calculation, defaults by 500. Fs is the sampling rate of input signal, defaults by 1.

The algorithm is as follows:

```
         [h(1),h(2),...h(NumSamp)] * conj([h(1),h(2),...h(NumSamp)].')
NBW = Fs * ------------------------------------------------------------
                 abs(sum([h(1),h(2),...h(NumSamp)])).^2
```

where h(1),h(2),... h(NumSamp) is impulse response of the given filter.

Examples

Compatibility

See also

num2str

Syntax
ystring = num2str(x)

Definition
This function can convert a real-valued scalar, vector or array to a string representation. Only real portions of complex valued numbers will be entered to the string. Arrays are traversed along the innermost (column) dimension. Commas, semicolons, brackets and other non-whitespace delimiters are ignored when drafting the string.

Examples:

Formula	Result
num2str(500)	'500'
num2str([500,200;100,400])	'500 100 200 400'
num2str(500+200i)	'500'

Compatibility
Real valued scalars, vectors, arrays

See Also
str2num (users)

numel

Syntax
y = numel(x)

Definition
This function returns the total number of parts in the array *x*.

Examples:

Formula	Result
numel(2)	1
numel([1 2 3])	3
numel(diag([1 1]))	4

Compatibility
scalars, vectors, arrays

oct2dec

Convert octal to decimal numbers

Syntax

d = oct2dec(c)

Definition

D = oct2dec(C) converts octal matrix C to a decimal matrix D, element by element. In both representations, the rightmost digit is the least significant.

Examples

Compatibility

See also
bi2de (users)

phasedelay

Phase delay vector of digital filter

Syntax

[phi,w] = phasedelay(b,a,n)

[phi,w] = phasedelay(b,a,n,'whole')

phi = phasedelay(b,a,w)

[phi,f] = phasedelay(b,a,n,fs)

[phi,f] = phasedelay(b,a,n,'whole',fs)

phi = phasedelay(b,a,f,fs)

Definition

1. [PHI,W] = phasedelay(B,A,N) returns frequency vector W and phase delay vector PHI of the filter defined by numerator coefficients of B and denominator coe -fficients of A. The length of PHI and W are N. W is n points equally spaced from 0 to pi. N should be integer larger than 1.
2. [PHI,W] = phasedelay(B,A,N,'whole') uses n points eually spaced from 0 to 2*pi
3. PHI = phasedelay(B,A,W) returns the phase delay at given frequencies specified in W. W is normally between 0 and pi.
4. [...] = phasedelay(...,fs) is same with above except the frequency vector is in HZ.
5. In [PHI,F] = phasedelay(B,A,N,FS), F is equally spaced from o to FS/2.
6. In [PHI,F] = phasedelay(B,A,N,'whole',FS), F is equally spaced from o to FS.
7. In PHI = phasedelay(B,A,F,FS), F should be in the range of 0 to FS/2.

⚠ Output arguments should **NOT** be omitted

Examples

Compatibility

poly2trellis

Convert convolutional code polynomials to trellis description

Syntax

Y = poly2trellis(CONSLEN,CODEGEN)

Definition

Y = poly2trellis(CONSLEN,CODEGEN) generates coding trellis of convolutional coder, from code constraint length and code generating polynomials.

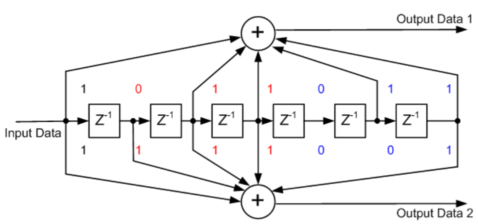
- **CONSLEN**: K by 1 vector, whrer K is the number of input bit streams to the encoder, or the number of bits the encoder takes simultaneously each clock cycle. CONSLEN[i]-1 specifys the number of registors used for the i'th input bit stream.

- **CODEGEN**: K by N matrix of octal numbers, where N is the number of output bit streams from the encoder, or the number of bits the encoder generates simultaneously each clock cycle. CODEGEN specifys the relationships between the K input streams and the N output streams.

- **Y**: 5-element struct, trellis description of the encoder:
  - numInputSymbols : equals to 2^K
  - numOutputSymbols: equals to 2^N
  - numStates : number of the registor states inside the encoder, equals to $2^{sum(CONSLEN-1)}$
  - nextStates : numStates by numInputSymbols matrix. nextStates(Row,Col) specifys the index of next state when the index of current state is Row and index of input symbol is Col.
  - outputs : numStates by numInputSymbols matrix. outputs(Row,Col) specifys the index of output symbol when the index of current state is Row and the index of input symbol is Col.

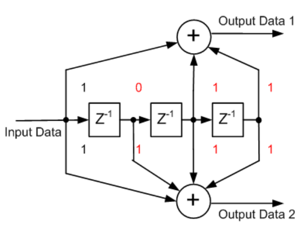
Examples

For the classic (2,1,6) code shown below, the parameters should be




```
ConsLen = [7]; % 7=m+1, m=6 is the number of registers
CodeGen = [133,171]; % octal 133=[1 0 1 1 0 1 1], octal 171=[1 1 1 1 0 0 1]
```

For the (2,1,3) code shown below, the parameters should be




```
ConsLen = [4]; % 4=m+1, m=3 is the number of registers
CodeGen = [13,17]; % octal 13=[1 0 1 1], octal 17=[1 1 1 1]
trellis = poly2trellis(ConsLen,CodeGen);
```

results

```
trellis =
 numInputSymbols: [2]
```

```
 numOutputSymbols: [4]
 numStates: [8]
 nextStates: [8x2 double]
 outputs: [8x2 double]
trellis.numInputSymbols =
 2
trellis.numOutputSymbols =
 4
trellis.numStates =
 8
trellis.nextStates =
 0  4
 0  4
 1  5
 1  5
 2  6
 2  6
 3  7
 3  7
trellis.outputs =
 0  3
 3  0
 3  0
 0  3
 1  2
 2  1
 2  1
 1  2
```

Compatibility

See also
convenc (users), *vitdec* (users)

true

Syntax
true

Definition
The value true (logical 1).

Examples:

Formula	Result
x = true	x is a true (logical 1)

See Also
false (users)

puncture

Erase specified symbols based on puncture pattern

Syntax

Y = puncture(X, puncPat)

Definition

puncPat: a vector of 1's and 0's, such as [1 0 0 1 1]

Examples

```
x = 1:10;
puncPat = [1 0 1 1];
y = puncture(x,puncPat),
y =
 1 3 4 5 7 8 9
```

Compatibility

See also
depuncture (users)

qamdemod

Quadrature amplitude demodulation

Syntax

Definition

Examples

Compatibility

No

See also
qammod (users)

qammod

Quadrature amplitude modulation

Syntax

Definition

Examples

Compatibility

See also
qamdemod (users)

qfunc

Q function

Syntax

Y = qfunc(X)

Definition

Y = qfunc(X) returns the Q function of real scalar X, i.e.

```
                1       _+Inf      t^2
Y = Q(X) = ----------  |     exp(- ---) dt
          sqrt(2*pi) _|X            2
```

The Q function is related to complementary error function *erfc* (users), according to

```
      1         X
Q(X) = - erfc( ----- )
      2      sqrt(2)
```

Examples

Compatibility

See also
qfuncinv (users), *erf* (users), *erfc* (users)

qfuncinv

Inverse Q function

Syntax

Y = qfuncinv(X)

Definition

Y = qfuncinv(X) returns the argument of Q function of real scalar X satisfying:

```
                 1        _+Inf       t^2
X = Q(Y) = ----------  |      exp(- ---) dt
          sqrt(2*pi) _|Y           2
```

The Q function is related to complementary error function ERFC, according to

```
       1         X
Q(X) = - erfc( ----- )
      2      sqrt(2)
```

Examples

Compatibility

See also
qfunc (users), *erf* (users), *erfc* (users)

rand

Syntax
y = rand(n1)
OR
y = rand(n1,n2)
OR
y = rand([n1,n2,..nN])

Definition
This function returns an array of random numbers with uniform distribution. The size of the array can be specified either as a list of one or two scalars or a vector for higher-dimensions. If a single scalar *n1* is used as the only parameter, a square matrix of size *n1* x *n1* is returned.

Examples:

```
% Create a 5x5 matrix of uniformly distributed random numbers
y = randn(5)
%
% Create a 5-part row vector of uniformly distributed random numbers
y = randn(1,5)
%
% Create a 3x4x2 matrix of uniformly distributed random numbers
y = randn([3,4,5])
%
```

Compatibility
nN - positive integer valued scalar or vector for all N >= 1.

See Also:
randn (users)

randerr

Generate bit error patterns

Syntax

OUT = randerr(M)

OUT = randerr(M,N)

OUT = randerr(M,N,ERRORS)

OUT = randerr(M,N,ERRORS,STATE)

Definition

OUT = RANDERR(M) generates an M-by-M binary matrix, each row of which has exactly one nonzero entry in a random position.

OUT = RANDERR(M,N) generates an M-by-N binary matrix, each row of which has exactly one nonzero entry in a random position.

OUT = RANDERR(M,N,ERRORS) generates an M-by-N binary matrix, where errors determines how many nonzero entries are in each row.

OUT = RANDERR(M,N,ERRORS,STATE) specifies the state of the random number generator.

Examples

Compatibility

See also
rand (users), *randsrc* (users), *randint* (users)

randint

Generate uniformly distributed random integers

Syntax

OUT = randint

OUT = randint(M)

OUT = randint(M,N)

OUT = randint(M,N,RANGE)

OUT = randint(M,NRANGE,STATE)

Definition

OUT = randint generates a random scalar that is either 0 or 1 with equal probability.

OUT = randint(M) generates a random M-by-M binary matrix,the elements of which take the value 0 or 1 with equal probability.

OUT = randint(M,N) generates random M-by-N binary matrix.

OUT = randint(M,N,RANGE) specifis the element value range. If RANGE is a positive integer, the OUT will be from [0, RANGE-1]. If RANGE is a negative integer, the OUT will be from [RANGE+1,0]. If RANGE is two element vector, the OUT will be from [MIN, MAX].

OUT = randint(M,NRANGE,STATE) specifies the state of the random number generator.

Examples

Compatibility

See also
rand (users), *randsrc* (users), *randerr* (users)

randn

Syntax
y = randn(n1)
OR
y = randn(n1,n2)
OR
y = randn([n1,n2,..nN])

Definition
This function returns an array of random numbers with Normal (Gaussian) distribution. The size of the array can be specified either as a list of one or two scalars or a vector for higher-dimensions. If a single scalar *n1* is used as the only parameter, a square matrix of size *n1* x *n1* is returned.

Examples:

```
% Create a 5x5 matrix of normally distributed random numbers
y = randn(5)
%
% Create a 5-part row vector of normally distributed random numbers
y = randn(1,5)
%
% Create a 3x4x2 matrix of normally distributed random numbers
y = randn([3,4,5])
%
```

Compatibility
nN - positive integer valued scalar or vector for all N >= 1.

See Also:
rand (users)

randsrc

Generate random matrix using prescribed alphabet

Syntax

OUT = randsrc

OUT = randsrc(M)

OUT = randsrc(M,N)

OUT = randsrc(M,N,ALPHABET)

OUT = randsrc(M,N,[ALPHABET;PROB])

OUT = randsrc(M,N,[ALPHABET;PROB],STATE)

Definition

OUT = randsrc generates a random scalar that is either -1 or 1 with equal probability.

OUT = randsrc(M) generates a random M-by-M matrix, the element of which is 1 or -1 with equal probability.

OUT = randsrc(M,N) generates a random M-by-N matrix.

OUT = randsrc(M,N,ALPHABET) specifies the ALPHABET instead of the default -1 and 1.

OUT = randsrc(M,N,[ALPHABET;PROB]) specifies the probabilty of each alphabet.

OUT = randsrc(M,N,[ALPHABET;PROB],STATE) specifies the state of the random number generator.

Examples

Compatibility

See also
rand (users), *randint* (users), *randerr* (users)

rcosflt

Filter input signal with (sqrt) raised cosine filter

Syntax

Y = rcosflt(X,OSR,fltType,Alpha)

Y = rcosflt(X,OSR,fltType,Alpha,Delay)

[Y,H] = rcosflt(X,OSR,fltType,Alpha,...)

Definition

1. Y = RCOSFILT(X,OSR,fltType,Alpha,Delay) filters the input signal vector with automatically designed raised cosine FIR filter, where

    X: Original input data (without zero inserted)
    OSR: Oversample ratio, the sample frequency ratio of output signal to input signal, must be a integer at least 1.
    fltType: "normal" or "sqrt", specifies the filter used is a normal raised cosine filter or root raised cosine filter, defaults by "normal".
    Alpha: Rolloff factor of the filter, or the ratio of extra bandwidth to input sample rate, must be in the range of [0,1], defaults by 0.5. If OSR is 1, Alpha can only be set 0.
    Delay: Group delay of the filter, measured in input samples, defaults by 8. The designed filter length is 2*OSR*Delay+1.

    Root mean square (rms) value of Y is almost the same as that of X. Length of output signal is (length(x)+2*Delay)*OSR when OSR is greater than 1. If X is a matrix, each column is treated as a independent signal.
2. [Y,H] = RCOSFILT(X,OSR,fltType,Alpha,...) returns the filter used in H.
3. This function is generally used for pulse shaping of baseband signal with the following process:
    1. Design a raised cosine filter based on OSR,fltType,Alpha and Delay.
    2. Zeros inserting (X) according to OSR,
    3. Filtering with designed filter.

Examples

Compatibility

See also
firrcos (users)

real

Syntax
y = real(x)

Definition
This function returns the real part of a complex number. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
real(20)	20
real(3+2j)	3
real([-2+4j 5-3j 2+j])	[-2 5 2]

Compatibility
Numeric scalars, vectors, arrays

See Also
imag (users)

rectpulse

Rectangular pulse shaping

Syntax

Y = rectpulse(X,nSamp)

Definition

Y = RECTPULSE(X,nSamp) return the rectangular pulse shaped signal of X by replicates each symbol in X nSamp times. If X is a matrix, each column is treated as a independent signal.

Examples

Compatibility

See also
upsample (users), *rcosflt* (users)

rectwin

Rectangular window

Syntax

C = rectwin(L)

Definition

C = rectwin(L) returns an all-1 column vector C of length L.

Examples:

Formula	Result
rectwin(5)	[1 1 1 1 1].'

See Also:
bartlett (users), *triang* (users), *blackman* (users), *gausswin* (users), *hamming* (users), *hann* (users)

rem

Syntax
r = rem(a,b)

Definition
This function returns the remainder when dividing *a* by *b*. Both parameters are required to be real arrays or real scalars subject to the restriction that if *b* is a vector or array, it must be the same size as *a* for part-by-part division and remainder computation. when *b* is a scalar, all the parts of *a* are divided by it. When *b* is explicitly zero, the result is *NaN*.

Examples:

Formula	Result
rem(2, 1.45)	0.55
rem([2,5,6], 1.45]	[0.55, 0.65, 0.20]
rem([2,5,6], [1.45,1.55,1.65]	[0.55, 0.35, 1.05]

Compatibility
Real valued scalars, vectors, arrays

See Also
mod (users)

resample

Change sampling rate by rational factor

Syntax

Y = resample(X,P,Q)

Y = resample(X,P,Q,N)

Y = resample(X,P,Q,N,Beta)

Y = resample(X,P,Q,B)

[Y,H] = resample(X,P,Q,...)

Definition

1. Y = RESAMPLE(X,P,Q) resamples the signal in vector X at P/Q times the original sample rate using a polyphase implementation. P and Q must be positive integers. Length of Y is ceil(length(x)*P/Q). If X is a matrix, resample works down its columns.
2. Resample applies an anti-aliasing (lowpass) FIR filter to X during the resampling process. It designs the filter using FIRLS with a Kaiser window. Cutoff frequency of the filter is 1/max(p,q). Signal delay introduced by filtering is compensated. Deviations may exist at both ends of Y due to filter transient (data preceding and following given data is regarded as zero).
3. Y = RESAMPLE(X,P,Q,N) uses 2*N*max(1,Q/P) samples of X to compute each sample of Y. The length of the FIR filter is 2*N*max(P,Q)+1; By default, N=10. If N = 0, RESAMPLE performs a nearest-neighbor interpolation, i.e., the output Y(k) = X(round((k-1)*Q/P)+1).
4. Y = RESAMPLE(X,P,Q,N,BETA) uses BETA as the BETA parameter for the Kaiser window. By default, BETA = 5.
5. Y = RESAMPLE(X,P,Q,B) equals to the following processes (regardless of filter transient): ceil(length(x)*p/q)

   ```
   Y1 = zeros(length(X)*P,1);
   Y1(1:P:end) = X;
   Y2 = CONV(X,B);
   Y = Y2((ceil((length(B)-1)/2)+(1:Q:1+Q*ceil(length(X)*P/Q)-Q)));
   ```
   B should be a odd length phase linear filter to keep Y(1) the same time index as X(1). If B is of even length, Y(1) is coordinate with X(1+1/P).

6. [Y,B] = RESAMPLE(X,P,Q,...) returns the vector B, the coefficients of the filter applied to x during the resampling process (after upsampling).

Examples

Compatibility

See also
downsample (users), *firls* (users), *interp* (users), *interp1* (users), *kaiser* (users), *upfirdn* (users), *upsample* (users)

reshape

Syntax
y = reshape(x , i,j)
y = reshape(x , i,j,k, ...)
y = reshape(x , [i,j,k, ...])
y = reshape(x , ...,[],...)

Definition
y = reshape(x , i,j) returns a i-by-j matrix with elements taken column wise from x. The number of elements in the resulting i-by-j matrix *y* must be same as number of elements in the input matrix *x*.
y = reshape(x , i,j,k, ...) and y = reshape(x , [i,j,k, ...]) will return a i-by-j-by-k-by.... matrix with same elements as in input matrix *x*. The number of elements in the resulting i-by-j-by-k-by.... matrix *y* must be same as number of elements in the input matrix *x*.
y = reshape(x , ...,[],...) replaces [] with an integer scalar number representing the number of elements in the corresponding dimension such that the total number of elements in output matrix *y* is same as the number of elements in input matrix *x*. You can have only one instance of [] in argument.

Swept-dimensions are NOT counted. (eg. if S is the variable produced by a 100 point linear analysis of a 2-port circuit, reshape(S, [4;1]) would return a variable containing S, but having dimensions 100x4x1)

Examples:

Formula	Result
x = [1 , 2 , 3 ; 4 , 5 ,6] y = reshape(x , 1, 6)	y = [1 , 4, 2, 5, 3 , 6]
x = [1 , 2 , 3 ; 4 , 5 ,6] y = reshape(x , [6, 1]) Or y = reshape(x , 6, 1)	y = [1; 4; 2; 5; 3; 6]
x = [1 , 2 , 3 ; 4 , 5 ,6 ; 7, 8, 9; 10, 11, 12] y = reshape(x , 6, [])	y = [1, 8; 4, 11; 7, 3; 10, 6; 2, 9; 5, 12]

Compatibility
Real and complex-valued Scalars, Vectors, Arrays

roots

Syntax
polyroot = roots(polycoef)

Definition
This function returns a column vector, *polyroot*, whose parts are the roots of the polynomial expressed in the form of the coefficient vector *polycoef*.

Examples:

```
% Find the roots of the polynomial:
% y = 1 - 6*x - 72*x^2 - 27*x^3
polycoef = [1,-6,-72,-27];
polyroot = roots(polycoef);
% polyroot = [12.1229;-5.7345;-0.3884]
```

Compatibility
Real or complex valued vector

round

Syntax
y = round(x)

Definition
round rounds the argument to the nearest integer. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
round(2.2)	2
round(2.2 + 3.7j)	2 + 4j
round(-2.3 - 3.9j)	-2 - 4j

Compatibility
Numeric scalars, Vectors, Arrays

See Also
floor (users)
ceil (users)
fix (users)

rsdec

Reed-Solomon decoder

Syntax

MSG = rsdec(CODE, M, K, PrimPoly)

MSG = rsdec(CODE, M, K, PrimPoly, B)

MSG = rsdec(CODE, M, K, PrimPoly, B, ErasLoc)

Definition

- **CODE**: received symbol block to be decoded, length within [2T+1,N], each symbol should be an integer within [0,2^M-1]
- **M**: the code is defined in GF(2^M), a message symbol represents M bits
- **K**: unshortened message length, an odd integer within [1,2^M-3]
- **PrimPoly**: P(x), primitive polynomial (Galois Field generator polynomial), M+1 terms with degree of M, highest degree item first
- **B**: degree of alpha, alpha^B is the first root of generator polynomial, B is 1 by default.
- **ErasLoc**: position vector of erasure symbols, each within [1,length(CODE)]

  If Ns = length(CODE) is less than N=2^M-1, N-Ns zeros shall be padded to head of CODE before decoding and discarded after decoding.

  Derived variables are:

- **N**: N=2^M-1, is the unshortened codeword length, such as 7, 15, 255, etc
- **a**: alpha, the prime element from which the Galois Field is generated, is commonly 02Hex in implementation.
- **T**: (N-K)/2, number of error symbols that can be corrected, 2*T+1 is the minimum distance between any two codewords
- **G(x)**: Galois Field generator polynomial,

$$G(x) = (x+a^{B}) * (x+a^{(B+1)}) * (x+a^{(B+2)}) * \ldots * (x+a^{(B+2T-1)})$$

code polynomial: C(X) = code(1:Ns) .* [$X^{(Ns-1)}$, $X^{(Ns-2)}$, ... X^1, x^0]

msg polynomial: MSG(X) = msg(1:Ks) .* [$X^{(Ks-1)}$, $X^{(Ks-2)}$, ... X^1, x^0]

Examples

```
% An R-S code in GF(2^4) correcting 3 errors can be coded/decoded as follows.
M = 4; % N = 2^M-1 = 15
K = 9; % T = 3, K = N-2*T
PrimPoly = [1,0,0,1,1] % P(x) = x^4 + x + 1
B = 1;
msg = [9,2,15,12,13,7,7,2,8]; % each symbol must in [0,2^M-1]
code = rsenc(msg,M,K,PrimPoly,B), % encoding
% code      = [9 2 15 12 13 7 7 2 8 14 12 12 6 8 12]; % [msg,parity]
code_error = [9 2 4 12 13 11 7 2 8 14 12 15 6 8 12]; % 3 error symbols
msg_dec = rsdec(code, M, K, PrimPoly, B), % decoding
% if some received symbols are known to be errors, erase decoding can be used
% the sum of error symbols and half the erased symbols should be not great than T
code_error_erase = [9 2 4 12 0 11 7 2 8 14 12 0 6 8 12]; % 2 error symbols and 2 erased symbols
ErasLoc = [4,12];
msg_dec_eras = rsdec(code, M, K, PrimPoly, B, ErasLoc), % erase decoding
```

results

```
msg_dec =
 9 2 15 12 13 7 7 2 8
msg_dec_eras =
 9 2 15 12 13 7 7 2 8
```

Compatibility

See also
rsenc (users)

rsenc

Reed-Solomon encoder

Syntax

CODE = rsenc(MSG, M, K, PrimPoly)

CODE = rsenc(MSG, M, K, PrimPoly, B)

Definition

- **MSG**: information symbol block to be encoded, length within [1,K], each symbol should be an integer within [0,2^M-1]
- **M**: the code is defined in GF(2^M), a message symbol represents M bits
- **K**: unshortened message length, an odd integer within [1,2^M-3]
- **PrimPoly**: P(x), primitive polynomial (Galois Field generator polynomial), M+1 terms with degree of M, highest degree item first
- **B**: degree of alpha, alpha^B is the first root of generator polynomial, B is 1 by default.

  If Ks = length(MSG) is less than K, K-Ks zeros shall be padded to head of MSG before encoding and discarded after encoding. The codeword is called shortened R-S code.

  Derived variables are:

- **N**: N=2^{M-1}, is the unshortened codeword length, such as 7, 15, 255, etc
- **a**: alpha, the prime element from which the Galois Field is generated, is commonly 02Hex in implementation.
- **T**: (N-K)/2, number of error symbols that can be corrected, 2*T+1 is the minimum distance between any two codewords
- **G(x)**: Galois Field generator polynomial,

  G(x) = (x+a^B) * (x+$a^{(B+1)}$) * (x+$a^{(B+2)}$) * ... * (x+$a^{(B+2T-1)}$)

  msg polynomial: MSG(X) = msg(1:Ks) .* [$X^{(Ks-1)}$, $X^{(Ks-2)}$, ... X^1, x^0]

  code polynomial: C(X) = code(1:Ns) .* [$X^{(Ns-1)}$, $X^{(Ns-2)}$, ... X^1, x^0]

  code(1:Ns) = [msg(1:Ks), parity(1:2T)]

Examples

```
% An R-S code in GF(2^4) correcting 3 errors can be coded as follows.
M = 4; % N = 2^M-1 = 15
K = 9; % T = 3, K = N-2*T
PrimPoly = [1,0,0,1,1] % P(x) = x^4 + x + 1
B = 1;
msg = [9,2,15,12,13,7,7,2,8]; % each symbol must in [0,2^M-1]
code = rsenc(msg,M,K,PrimPoly,B), % encoding, [msg,parity]
```

results

```
code =
 9 2 15 12 13 7 7 2 8 14 12 12 6 8 12
```

Compatibility

See also
rsdec (users)

runanalysis

Syntax
runanalysis('AnalysisName')
runanalysis('AnalysisName', ContinueOnError)

Definition
The runanalysis function is used to force an analysis to run from an equation block. It can be used to control simulations in a sequential manner. The function does not return until the analysis finishes, whether successful or in error.

The second argument, ContinueOnError, is optional and defaults to false. If ContinueOnError is false and an error is encountered when running the analysis, the equation block throws an error and terminates. If ContinueOnError is true, the equation script continues to run.

Examples:

```
SourceAmpls = [1 2 5 10];  % We'll step our source's amplitude with these values
for i = 1 : length(SourceAmpls)
 CurAmplitude = SourceAmpls(i);  % This variable is used by our source's Amplitude parameter
 runanalysis('Analysis1');
 % Post process data from the current analysis run
 % Post-processing equations would go here
end
```

Compatibility

See Also

sec

Syntax
y = sec(x)

Definition
sec returns the secant of a radian-valued argument. This function operates on an part-by-

part basis on arrays.

Compatibility
Numeric scalars, Vectors, Arrays

secd

Syntax
y = secd(x)

Definition
secd returns the secant of a degree-valued argument. This function operates on an part-by-part basis on arrays.

Compatibility
Numeric scalars, Vectors, Arrays

sech

Syntax
y = sech(x)

Definition
sech returns the hyperbolic secant the argument. This function operates on an part-by-part basis on arrays.

Compatibility
Numeric scalars, Vectors, Arrays

setindep

Syntax
setindep("dependentvar", "independentvar1", "independentvar2", ...)

Definition
setindep manually sets the independent variable(s) for a swept variable. Both are passed by name. A long name can be used for the independentvar. If independentvar is empty (blank) the dependentvar becomes unswept. All independents should have the same length, equal to the number of rows in the dependent.

Examples:

Formula	Result
ind = [0.025;1;2;5] setindep("x" ,"ind")	set x to have a 4 part independent vector. x should be of size 4xm or 4xmxn
abest = myS[2,1] setindep("abest", "myData.F")	set abest to use MyData.F as an independent vector. F must have the same number of parts as abest has rows.

Compatibility
Vectors and Arrays. The independent var must be numeric.

See Also
getunits

setmatlabvariables

Syntax
setmatlabvariables('var1', 'var2')
setmatlabvariables var1 var2

Definition
setmatlabvariables sets SystemVue's variables to MATLAB, which will defines MATLAB variables and set value to variable

Examples:

Formula	Result
setmatlabvariables('var1', 'var2')	set two variables named var1/var2 in the SystemVue to MATLAB

Compatibility
variables are strings.

See Also
MATLAB Integration, *getmatlabvariables* (users)

setunits

Syntax
setunits('varname', unit)

Definition
setunits sets a variable named varname to have units specified by the parameter unit. unit may be an integer or a string.

setunits is used only to set the units of variables in equations and datasets. It will not change units of a part's parameters.

Examples:

Formula	Result
y = [0.025] setunits('x' , 6006)	sets units of y to um y = 25000
y = 5 setunit('y' , 'mm')	sets units of y to mm y =5000
y = 0.0001 setunits('y', 'uF')	sets units of y to uF y = 100

Compatibility
Numeric Scalars, Strings

See Also
getunits

setvariable

Syntax
setvariable(Dataset, Variable, value)

Definition
setvariable sets a variable value in a dataset

Examples:

Formula	Result
setvariable('OutData', 'OutVar', 3)	set the variable named Outvar in the dataset OutData to the value 3
setvariable('Out', 'Var', [1 2 3])	set the variable named Var in the dataset Out to a vector [1 2 3]

Compatibility
Dataset and Variable are strings. value is any valid value.

See Also
getvariable (users)

sftrans

transform of lowpass filter to other type filter

Syntax

[fz, fp, fg] = sftrans(z,p,g,w,stop)

Definition

1. This function transform the zero-pole-gain of a lowpass filter with normalized bandwidth to lowpass filter, highpass filter, bandpass filter or bandstop filter. W is

desired bandwidth which for lowpass and highpass has one element,and for bandpass and bandstop has two elements [W1 W2]. STOP is set true for highpass and bandstop filter or set false for lowpass and bandpass filter. W is in s-plane and is in rad/s.

2. ⚠ Output arguments should NOT be omitted

Examples

Compatibility

See also

sign

Syntax
y = sign(x)

Definition
sign returns the signum of the argument. The signum function returns -1 if the argument is negative, 1 if the argument is positive, and 0 if the argument is 0. This function operates on an part-by-part basis on arrays.

Compatibility
Numeric scalars, Vectors, Arrays

sin

Syntax
y = sin(x)

Definition
sin returns the sine of the radian-valued argument. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
sin(0)	0
sin(pi/2)	1
sin(-pi/2)	-1
sin([pi/4 2*pi/3])	[0.707 0.866]

Compatibility
Numeric scalars, Vectors, Arrays

See Also
asin (users)
sind (users)

sinc

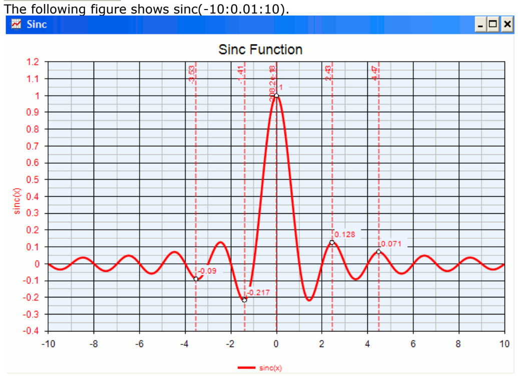
Syntax
y = sinc(x)

Definition
sinc returns the sinc function of the argument. The sinc function is defined as sin(pi*x)/(pi*x) or 1 if x is equal to 0. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
sinc(0)	1
sinc(pi/2)	0.198
sinc(pi/4)	0.253
sinc(2*pi/3)	0.044

The following figure shows sinc(-10:0.01:10).



Compatibility
Numeric scalars, Vectors, Arrays

See Also
sin (users)

sind

Syntax
y = sind(x)

Definition
sind returns the sine of the degree-valued argument. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
sind(0)	0
sind(90)	1
sind(-90)	-1
sind([45 60])	[0.707 0.866]

Compatibility
Numeric scalars, Vectors, Arrays

See Also
asin (users)
sin (users)

sinh

Syntax
y = sinh(x)

Definition
sinh returns the hyperbolic sine of the number, or (exp(x) - exp(-x)) / 2. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
sinh(1)	1.175
sinh(5)	74.203
sinh([pi/3 0])	[1.249 0]

Compatibility
Numeric scalars, Vectors, Arrays

See Also
asinh (users)

size

Syntax
y = size(x)

Definition
size returns a vector containing the number of parts in each dimension of x. part one of y corresponds to the number of parts in the first dimension, part two to the second dimension, and so on.

Examples:

Formula	Result
size([1 2 3 4])	[1 4]
size([1 2 3; 4 5 6])	[2 3]
size(ones(4,3,2))	[4 3 2]

Compatibility
Numeric Scalars, Vectors, Arrays

skewness

Syntax
y = skewness(x)
y = skewness(x,Flag)
y = skewness(x,Flag,iDim)

Definition
Returns the sample skewness of a vector x. Skewness is the third central moment of X divided by the cube of the standard deviation.

If Flag is 0 (default), skewness normalizes by N-1 where N is the sample size. If Flag is 1, skewness normalizes by N.

For matrices, this function operates separately on each column and returns a vector. For multi-dimensional arrays in general, this function operates on the dimension specified by iDim, or the first non-singleton dimension if iDim is not specified.

Examples:

Formula	Result
y = skewness([3 ; 4 ; 8 ; 9])	y = 0
y = skewness([1, 2, -5], 1)	y = -0.652

Compatibility
Numeric arrays

See Also
std (users)
var (users)

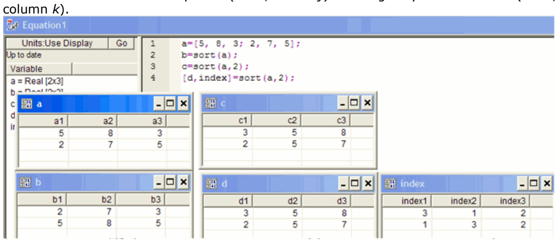
sort

Syntax
y = sort(x)
y = sort(x,dim)
[y,index]=sort(x)
[y,index]=sort(x,dim)

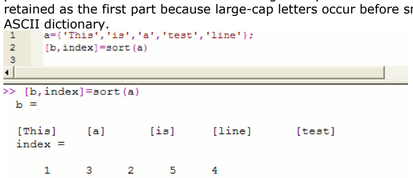
Definition
This function sorts contents of the array x in ascending order along one specific dimension of the array. When unspecified, the innermost non-singleton dimension is chosen. The function can be required to additionally specify the original indices in the sorted order.

Examples:

In the following example note that **b** is the column-wise (default dim is 1 for a 2x3 matrix) sorted whereas **c** and **d** are sorted row-wise. The **index** matrix associated with **d** is interpreted as follows: if the value *k* appears at a specific location along row *i* column *j*, it means that the number now placed (row *i*,column *j*) was originally the number at (row *i*, column *k*).



Strings can be sorted alphabetically according to ASCII dictionary if the collection is presented as cells as shown in the following example. Note that here the string "This" is retained as the first part because large-cap letters occur before small-cap letters in the ASCII dictionary.



Compatibility:
Real-valued numeric vectors and arrays or strings

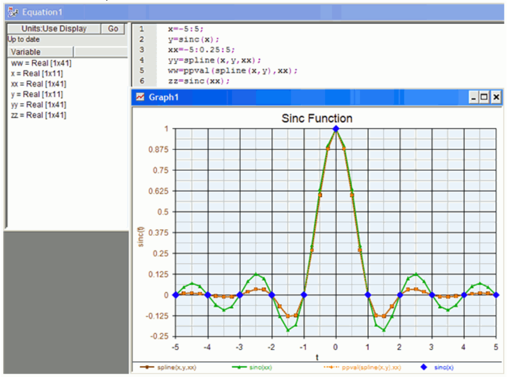
spline

Syntax
polynomial = spline(originalIndep,originalDep)
OR
fittedDependent = spline(originalIndep,originalDep,fittedIndep)

Definition
This function performs spline polynomial extraction from a one-dimensional function defined as the mapping of an original independent vector onto an original dependent vector. If supplied with a third argument explicitly specifying the independent vector to which fitting is required, the function returns the fitted dependent vector. If the third

parameter is not supplied then a structure describing the piece-wise polynomial function is returned, which may then be used in a call to the ppval(polynomial,fittedIndep) function to generate the fittedDependent variable.

Examples:
In the following example, the original mapping of x and sinc(x) are shown in sparsely spaced blue dots, one dot per unit along the independent axis. When four times as much granularity is required, an extended fitting vector xx is introduced. Spline curves produced using this extended independent vector are compared against the true sinc() function of the extended vector. Note how there is substantial match when some variation is present in the original data, e.g. just one non-zero data point in the original dependent vector. In regions where there is absolutely no off-axis data in the dependent vector i.e. in the side-lobes, the spline() function is still able to partially recover the existence of the side lobes, if not the full amplitude of each.



Compatibility
Real-valued 1-dimensional vector: originalIndep, fittedIndep
Real or complex-valued array: originalDep

sqrt

Syntax
y = sqrt(x)

Definition
This function returns the square-root of the argument.
This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
sqrt(0)	0
sqrt(4)	2
sqrt(2+3i)	1.67415+j0.895977
sqrt(-1)	j
sqrt([9 16 -4])	[3 4 -2j]

Compatibility
Real and complex-valued scalars, vectors, arrays

square

Square wave generation

Syntax

S = square(Rad)

S = square(Rad,Duty)

Definition

1. S = SQUARE(Rad) generates a 50% duty square wave with period 2*Pi for the vector Rad (radian). Rad is the product of 2*PI, frequency and time.
2. S = SQUARE(T,DUTY) generates a square wave with specified duty cycle. Duty is the percent of the period in which the signal is positive.

Examples

```
f = 100;
t = 0:.0001:.0100;
y = square(2*pi*f*t);
```

Compatibility

See also
cos (users), *sin* (users), *sinc* (users)

ss2tf

Convert state-space filter parameters to transfer function form

Syntax

[num,den] = ss2tf(a, b, c, d)

Definition

1. [NUM, DEN] = ss2tf(A, B, C, D). A should be square matrix. B should be column vector with length equal with A's column number. C should be row vector with length equal with B. D should be a scalar.

Examples

Compatibility

See also
ss2zp (users), *tf2ss* (users), *zp2tf* (users)

ss2zp

Convert state-space filter parameters to zero-pole-gain form

Syntax

[z, p, k] = ss2zp(a, b, c, d)

Definition

1. [Z, P, K] = ss2zp(A, B, C, D) A should be square matrix. B should be column vector with length equal with A's column number. C should be row vector with length equal with B. D should be a scalar.

Examples

Compatibility

See also
ss2tf (users), *tf2zp* (users), *zp2ss* (users)

sscanf

Syntax:
A = sscanf(string, format)
A = sscanf(string, format, size)
[A, count, msg, next] = sscanf(...)

Definition:
Used to read formatted input from a string. Converts the input string using format argument (format) and puts the results into a matrix (A).

size (optional) argument is used to determine how much data is read. Valid values are:

n	read at most n fields from the string
inf	read all of the input string
[m,n]	read at most m*n fields. Fill a matrix with at most m rows.

count (optional) result is the number of matching fields.
msg (optional) is for an error message
next (optional) is one more than the number of characters match in the input string

Format:

- **Whitespace characters** (space, tab or new lines) are used to delimit fields. There are not included in the output.
- **Non-whitespace characters** that are not a part of a format specifier are matched with the next character in string and then discarded. If the character does not match sscanf stops process string.
- **Format specifers**: %[*][width][modifiers]conversionChar, where:

*	(optional) match the data in string but do not put the corresponding match in the output matrix. The format must match but it isn't included in the output.
width	(optional) maximum number of characters to match in string
modifiers	(optional) For compatibility only. valid values (h, l, L)
conversionChar	see table below

Conversion Characters:

Type	Qualifying Input
c	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.
d	Decimal integer: Number optionally preceeded with a + or - sign.
e,E,f,g,G	Floating point: Decimal number containing a decimal point, optionally preceeded by a + or - sign and optionally folowed by the e or E character and a decimal number.
o	Octal integer.
s	String of characters. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).
u	Unsigned decimal integer.
x,X	Hexadecimal integer.

std

Syntax
y = std(x)
y = std(x, Flag)
y = std(x, Flag, iDim)

Definition
Returns the standard deviation of a vector x.

If Flag is 0 (default), std normalizes by N-1 where N is the sample size. If Flag is 1, std normalizes by N.

For matrices, this function operates separately on each column and returns a vector. For multi-dimensional arrays in general, this function operates on the dimension specified by iDim, or the first non-singleton dimension if iDim is not specified.

Examples:

Formula	Result
y = std([3 ; 4 ; 8 ; 9])	y = 2.9439
y = std([1, 2, 3], 1)	y = 0.8165

Compatibility
Numeric arrays

See Also
var (users)
skewness (users)

str2num

Syntax
y = str2num('xstring')

Definition
This function can convert a single real-valued number from string format to numeric format.
When supplied with a string containing preceeding non-numeric characters, other than whitespace or tab, the function returns zero.

Examples:

Formula	Result
str2num('500')	500
str2num(' 500')	500

Compatibility
String

See Also
num2str (users)

strcmp

Syntax
out = strcmp(str1, str2)
out = strcmp(str, ca)
out = strcmp(ca1, ca2)

Definition
out = strcmp(str1, str2) compares two strings, str1 and str2, and returns true (logical 1) if they are identical. If not, then it returns false (logical 0).

out = strcmp(str, ca) compares str with each string in a cell array. It then returns a logical array, out, that contains the corresponding logical values on whether the two strings are identical.

out = strcmp(ca1, ca2) compares each part in ca1 to the corresponding part in ca2. It then returns a character array that is the same size as ca1 and ca2 with the corresponding logical value on whether the two strings are identical.

This function does not ignore case. To ignore case, use the strcmpi function.

Examples:

Formula	Result
out = strcmp('One', 'Two')	out = 0
out = strcmp('Yes', {'No', 'Yes'})	out = [0, 1]

Compatibility
string array, cell array

See Also
strcmpi (users)

strcmpi

Syntax
out = strcmpi(str1, str2)
out = strcmpi(str, ca)
out = strcmpi(ca1, ca2)

Definition
out = strcmpi(str1, str2) compares two strings, str1 and str2, and returns true (logical 1) if they are identical. If not, then it returns false (logical 0).

out = strcmpi(str, ca) compares str with each string in a cell array. It then returns a logical array, out, that contains the corresponding logical values on whether the two strings are identical.

out = strcmpi(ca1, ca2) compares each part in ca1 to the corresponding part in ca2. It then returns a character array that is the same size as ca1 and ca2 with the corresponding logical value on whether the two strings are identical.

This function ignores case. To take the case into account, use the strcmp function.

Examples:

Formula	Result
out = strcmpi('One', 'Two')	out = 0
out = strcmpi('Yes', {'No', 'YES'})	out = [0, 1]

Compatibility
string array, cell array

See Also
strcmp (users)

strncmp

Syntax
out = strncmp(str1, str2, n)
out = strncmp(str, ca, n)
out = strncmp(ca1, ca2, n)

Definition
This function compares the first n characters in str1 and str2 and if they are identical, it returns true (logical 1). Otherwise, it returns false (logical 0).

The function can also compare a string and each part in a cell array, or the parts in two cell arrays.

This function is case sensitive. To ignore case, use the strncmpi function.

Examples:

Formula	Result
out = strncmp('example', 'exam', 4)	out = 1;
out = strncmp('test', {'exam', 'testing'}, 4)	out = [0,1];

Compatibility
string array, cell array

See Also
strncmpi (users)

strncmpi

Syntax
out = strncmpi(str1, str2, n)
out = strncmpi(str, ca, n)
out = strncmpi(ca1, ca2, n)

Definition
This function compares the first n characters in str1 and str2 and if they are identical, it returns true (logical 1). Otherwise, it returns false (logical 0).

The function can also compare a string and each part in a cell array, or the parts in two cell arrays.

This function is not case sensitive. To take case into account, use the strncmp function.

Examples:

Formula	Result
out = strncmpi('example', 'EXAM', 4)	out = 1;
out = strncmpi('test', {'exam', 'TeStING'}, 4)	out = [0,1];

Compatibility
string array, cell array

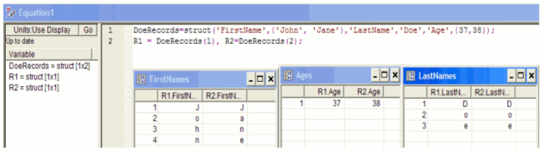
See Also
strncmp (users)

struct

Syntax
y = struct(field1,value1,filed2,value2,....,fieldN,valueN)

Definition
This function creates a structure parts of which can be of various types ranging from strings through complex cell arrays. Each field is assigned the type of the value which succeeds it. If the structure contains more than one cell array, like a matrix, all such cell arrays must be of the same size. Note that fields are always specified as strings.

Examples:
In the figure below, observe how records of two people who share the same last name can be saved to and retrieved from a single structure.



Compatibility
Numeric scalars, Vectors, Arrays

sum

Syntax
y = sum(x)
y = sum(x,dim)

Definition
Returns the sum of parts of a vector x.

For matrices, this function operates separately on each column and returns a vector. For multi-dimensional arrays in general, this function operates on the dimension specified by dim, or the first non-singleton dimension if dim is not specified.

The dim argument is optional and specifies which dimension to operate along. For example, if dim is 1, this function operates on each column of the argument. If the argument is omitted, the first non-singleton dimension is chosen as the dimension to operate along.

Examples:

Formula	Result
y = sum([10 , 3 , 5])	y = 18
y = sum([2 ; 9 ; 11])	y = 22
y = sum([complex(3 , 3) , complex(5 , 2)])	y = 8 + j5
y = sum([3 , 2 , 19 ; 5 , 7 , 1.5])	y = [8, 9, 20.5]
y = sum([3 , 2 , 19 ; 5 , 7 , 1.5], 2)	y = [24; 13.5]

Compatibility
Numeric scalars, Vectors, Arrays

svd

Syntax
s = svd(X)
[U,S,V] = svd(X)
[U,S,V] = svd(X, 0)
[U,S,V] = svd(X, 'econ')

Definition
Let X be an m x n matrix and k = min(m,n).

S = svd(X) returns, in the vector S, the singular values (in decreasing order) of the matrix X. S is a column vector of size k.

[U,S,V] = svd(X) produces matrices U, S, and V that form the singular value decomposition of X, that is, X = U·S·V', where
U is a unitary m x m matrix
S is a diagonal m x n matrix whose primary diagonal parts are the singular values (in decreasing order) of X
V is a unitary n x n matrix

[U,S,V] = svd(X, 0) OR [U,S,V] = svd(X, 'econ') produce matrices U, S, and V that form the 'economical' singular value decomposition of X, that is, X = U·S·V', where
U is an m x k matrix containing only the first k columns of the unitary matrix U returned by [U,S,V] = svd(X)
S is a diagonal k x k matrix whose primary diagonal parts are the singular values (in decreasing order) of X
V is an n x k matrix containing only the first k columns of the unitary matrix V returned by [U,S,V] = svd(X)

Examples:

```
>> X = [ 0.60099 0.766416 0.440156; 0.12712 0.857445 0.130142; 0.94683 0.447486 0.559306 ]
 X =
     0.60099   0.766416   0.440156
     0.12712   0.857445   0.130142
     0.94683   0.447486   0.559306
>> S = svd(X)
 S =
     1.6967
     0.663471
     0.0347664
>> [U,S,V]=svd(X)
 U =
    -0.628061   -0.11714    -0.769297
    -0.41523    -0.78565     0.458627
    -0.658122    0.607481    0.444796
 S =
     1.6967          0           0
          0   0.663471           0
          0          0   0.0347664
 V =
    -0.620837    0.610289    0.492046
    -0.667115   -0.740937    0.0772603
    -0.411726    0.280286   -0.867134
>> X = [0.723014 0.179696 -0.861837 0.441228; -0.328791 0.934672 1.68603 0.955427]
 X =
     0.723014   0.179696  -0.861837   0.441228
    -0.328791   0.934672   1.68603    0.955427
>> [U,S,V]=svd(X)
 U =
    -0.293781  0.955873
     0.955873  0.293781
 S =
     2.25294        0  0  0
           0  1.07425  0  0
 V =
    -0.233779   0.553425   0.721934  -0.343335
     0.373129   0.415505  -0.509149  -0.654903
     0.827729  -0.305779   0.463201  -0.0825176
     0.347831   0.653893  -0.0708765  0.668142
>> [U,S,V]=svd(X,'econ')
 U =
    -0.293781  0.955873
     0.955873  0.293781
 S =
     2.25294        0
           0  1.07425
 V =
    -0.233779   0.553425
     0.373129   0.415505
     0.827729  -0.305779
     0.347831   0.653893
```

symerr

Compute number of symbol errors and symbol error rate

Syntax

[number,ratio] = symerr(x,y)

[number,ratio] = symerr(x,y,flg)

[number,ratio,loc] = symerr(...)

[NUMBER,RATIO,LOC] = symerr(X,Y,FLG)

Definition

This function compares the symbol difference between X and those in Y.

If X and Y are of the same size, FLG may be 'overall','row-wise' and 'column - wise'. When FLG is 'overall', NUMBER and RATIO are scalar which mean the difference number and rate of all elements in X compared with those in Y. When FLG is 'row-wise', NUMBER and RATIO are column vectors which mean the difference number and rate of each row of X compared with that in Y. When FLG is 'column-wise',NUMBER and RATIO are row vectors which mean the difference number and rate of each column of X compared with that in Y. LOC is the same size with X, in which 0 means same, 1 means difference. Default is 'overall' in this case.

If X is MX-1 vector and Y is MX-NY matrix, FLG may be 'overall' and 'column-wise'. Default is 'overall'. In this case, X is extended to MX-NY matrix in which each column is same. Then the calculation is same with that when X and Y are of the same size.

If X is 1-NX vector and Y is MY-NX matrix, FLG may be 'overall' and 'row-wise'. Default is 'overall'. In this case, X is extended to MY-NX matrix in which each row is same. Then the calculation is same with that when X and Y are of the same size.

If Y is vector while X is matrix, Y will be extended to matrix in the same way.

Examples

Compatibility

tan

Syntax
y = tan(x)

Definition
tan returns the tangent of the radian-valued argument. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
tan(pi)	0
tan(pi/4)	1
tan(-pi/4)	-1
tan([5*pi/11 -5*pi/11])	[6.955 -6.955]

Compatibility
Numeric scalars, Vectors, Arrays

See Also
atan (users)
tand (users)

tand

Syntax
y = tand(x)

Definition
tand returns the tangent of the degree-valued argument. This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
tand(180)	0
tand(45)	1
tand(-45)	-1
tand([180 45])	[0 1]

Compatibility
Numeric scalars, Vectors, Arrays

See Also
atan (users)
tan (users)

tanh

Syntax
y = tanh(x)

Definition
tanh returns the hyperbolic tangent of the argument, defined as (exp(x) - 1) / (exp(x) + 1). This function operates on an part-by-part basis on arrays.

Examples:

Formula	Result
tanh(1)	0.762
tanh(5)	1
tanh(pi/3)	0.781
tanh([pi/6 0])	[0.48 0]

Compatibility
Numeric scalars, Vectors, Arrays

See Also
atanh (users)

tcpip

Syntax
t = tcpip(ipAddr, nPort)

Definition
tcpip creates a class object to do tcpip i/o over a lan. ipAddr is a string with the IP Address in dotted format, and nPort is a port number for the connection. Once created, use fopen, fwrite, fread, fprintf, fscanf, fclose to manipulate the port.

Examples:

Formula	Result
t = tcpip('127.0.0.1', 80)	Create an object to connect to the web server on this computer (port 80 on 'this')

Compatibility
TCP/IP connections via LAN. ipAddr is a char array, and nPort is an integer.

tcpip Properties
Modify the way the tcpip link works by setting properties in the created class object. tcpip supports the following properties

Property	Description
LocalHost	Local host descriptor
LocalPort	Local port descriptor
LocalPortMode	Specify automatic local port assignment
ReadAsyncMode	Specfiy whether an asynchronous read operation.
RemoteHost	The remote host ip address (char array)
RemotePort	The remote port # (integer)
Terminator	Terminator string, such as 'CR/LF'. ASCII value 0 - 127, or 'CR', 'LF', 'CR/LF', or 'LF/CR'
TransferDelay	Specifies whether or not to use Nagle's algorithm.
InputBufferSize	Size of the input buffer in bytes.
OutputBufferSize	Size of the output buffer in bytes.
Timeout	Time to wait before timing out on receive (in seconds, floating point).

tf2ss

Convert transfer function filter parameters to state-space form

Syntax

[a, b, c, d] = tf2ss(num, den)

Definition

1. [A, B, C, D] = tf2ss(NUM, DEN). A,B,C and D are returned state-space. NUM should be empty or a vector while DEN should be a vector longer than NUM.

Examples

Compatibility

See also
ss2tf (users), *tf2zp* (users), *zp2ss* (users)

tf2zp

convert transfer function filter parameters to zero-pole-gain form

Syntax

[z, p, k] = tf2zp(num, den)

Definition

1. [Z, P, K] = tf2zp(NUM, DEN). Z and P are column vectors, NUM should be empty or a vector while DEN should be a vector longer than NUM.

Examples

Compatibility

See also
ss2zp (users), *tf2ss* (users), *zp2tf* (users)

toeplitz

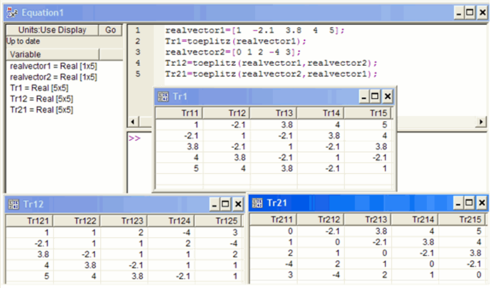
Syntax
tm = toeplitz(x)
OR
tm = toeplitz(x,y)

Definition
This function returns an m x m Toeplitz matrix based on an m-length vector x or a combination of m-length vectors x and y.

When only a single vector is used, the result is a symmetric, Hermitian matrix as shown in the Tr1 table below. Note that the vector parts are distributed symmetrically with respect to the principal dialoginal which is occupied by the first part of the input vector.

When two vectors are present, the first part of the first vector populates the principal diagonal as evidenced in the differences between Tr12 and Tr21. The other parts of the first vector populate the lower-triangle whereas those of the second vector populate the upper-triangle of the resultant matrix.

Examples:



triang

triangular window

Syntax

W = triang(N)

Definition

1. W = triang(N) returns the triangular window coefficients of length N in column vector W.

```
for N odd:
 W(k) = 2*k/(N+1),        if 1<= k <=(N+1)/2
      = 2*(N-k+1)/(N+1),  if (N+1)/2< k <=N
for N even:
 W(k) = 2*k/N,            if 1<= k <=(N+1)/2
      = 2*(N-k+1)/N,      if N/2+1< k <=N
```

Examples

Compatibility

See also
rectwin (users)

turbodec

Turbo decoder

Syntax

y=turbodec(x, g1, g2, map, puncture, tail, niter, algorithm, EbN0, rate)

Definition
This function decodes the codeword defined from *turboenc* (users)

Examples

Compatibility

See also
turboenc (users)

turboenc

turbo encoder

Syntax

y = turboenc(x, g1, g2, map, puncture, tail)

Definition

This function eccodes the input message with turbo generation polynomial defined below.

- **g1** and **g2** are binary form component generator each contains two rows the first is FeedbackPolynomial and the second is GeneratorPolynomial, e.g.

  g1 =

  [1 1 1;

  1 0 1]

- **map** is used as interleaver and deinterleaver. when interleaving, y(k)=x(map(k)), when deinterleaving, y(map(k))=x(k).
- If **puncture** = 1, coding rate is 1/3. If puncture = 0, coding rate is 1/2. when puncturing, odd check bits from component coder1 and even check bits from component coder2 are transmitted.
- If **tail** = 1, zerotailing bits of both component coder are transmitted. If tail = 0, zerotailing bits are not transmitted.

  Both component coder will be reset to zero at the beginning of a frame whether tail is 0 or 1.

Examples

Compatibility

See also
turbodec (users)

upfirdn

Upsample by zero inserting, filtering and downsampling a signal

Syntax

Y = upfirdn(X,H)

Y = upfirdn(X,H,P)

Y = upfirdn(X,H,P,Q)

Definition

1. Y = upfirdn(X,H,P,Q) performs a cascade of three operations:
   - Upsampling by the ratio of positive integer P (zero insertion). P defaults to 1.
   - FIR filtering the upsampled signal with impulse response sequence given in H.
   - Downsampling the filtered signal by the ratio of positive integer Q. Q defaults to 1.
2. If X and H are vectors, the output is also a vector whose size satisfies length(y) = ceil(((length(x)-1)*P + length(h))/Q). In this case, upfirdn(X,H,P,Q) results in the same results as the following procedure:

```
x_zeroInsrt = zeros(length(x)*p-p+1,1);
x_zeroInsrt(1:p:end) = x;
x_applyH = conv(h,x_zeroInsrt);
x_dnsmpl = x_applyH(1:q:(length(y)));
```

If X is a matrix and H is a vector, each column of X is filtered by H.
If X is a vector and H is a matrix, each column of H is used to filter a copy of X.
If X is a matrix and H is a matrix with the same number of columns, then the the i-th column of X is filered by the i-th column of H. If each column of X is identical, it's degraded to the case where X is a vector and H is a matrix.
Followed are the valid combinations of arguments.

X	H	Y
row(column) vector	vector	row(column) vector
matrix	vector	matrix
vector	matrix	matrix
matrix	matrix	matrix

Examples

Compatibility

See also
conv (users), *downsample* (users), *filter* (users), *interp* (users), *resample* (users), *upsample* (users)

upsample

Upsample input signal by inserting R-1 zeros between elements

Syntax

Y = upsample(X,R)

Y = upsample(X,R,OFFSET)

Definition

1. Y = upsample(X,R) upsamples input signal X by inserting R-1 zeros behind each input sample. X may be a vector or a matrix (one signal per column). For matrix, upsampling is applied to each column respectively.
2. Y = upsample(X,R,OFFSET) specifies an optional sample offset. OFFSET should be an positive integer within [0,R-1] and is 0 by default.

Examples

```
x = [1 2 3 4 5].';
y = upsample(x, 4);
z = upsample(x, 4, 1);
p = [1 0 0 0 2 0 0 0 3 0 0 0 4 0 0 0 5 0 0 0].'; % p equals to y
q = [0 1 0 0 0 2 0 0 0 3 0 0 0 4 0 0 0 5 0 0].'; % q equals to z
```

Compatibility

See also

downsample (users), *interp* (users), *interp1* (users), *resample* (users), *upfirdn* (users)

using

Syntax
using('DatasetName');

Definition
This function sets the current context in an equation block to the named dataset. When set, you can use the variables within the dataset as if there were defined in the equation block. This function can be used to context switch between datasets in any post processing Equation page.

Examples:
If there are two datasets, called "Data1" and "Data2" which both contain a variable called "Var1".
Then the way to access these variables without confusion is as follows:

```
using('Data1');
% Assume "Var1" of "Data1" is [3, 6, 9, 12]
z1=Var1/3;
using('Data2');
% Assume "Var1" of "Data2" is [2, 4, 6]
z2=Var1/2;
```

The results are:

```
z1 = [1, 2, 3, 4]
Z2 = [1, 2, 3]
```

Compatibility
String

var

Syntax
y = var(x)
y = var(x, W)
y = var(x, W, iDim)

Definition
Returns the variance of a vector x.

If W is 0 (default), var normalizes by N-1 where N is the sample size. If W is 1, var normalizes by N. If W is a vector, it is treated as coefficient weights for computing the variance. In this case, the coefficients of W are scaled so that they sum to unity.

For matrices, this function operates separately on each column and returns a vector. For multi-dimensional arrays in general, this function operates on the dimension specified by iDim, or the first non-singleton dimension if iDim is not specified.

Examples:

Formula	Result
y = var([3 ; 4 ; 8 ; 9])	y = 8.6667
y = var([1, 2, 3], 1)	y = 0.6667
y = var([1, 2, 3], [0.7, 0.1, 0.2])	y = 0.65

Compatibility
Numeric arrays

See Also
std (users)
skewness (users)

vitdec

Convolutionally decodes binary stream using Viterbi algorithm

Syntax

Y = vitdec(X,TRELLIS,tbLen,MODE,inType)

Y = vitdec(X,TRELLIS,tbLen,MODE,inType,puncPat)

Y = vitdec(X,TRELLIS,tbLen,'cont',inType,puncPat,initState)

[Y,finalState] = vitdec(X,TRELLIS tbLen,'cont',inType, ...)

Definition

Y = vitdec(X,TRELLIS,tbLen,MODE,inType,puncPat) decodes the input vector X using the Viterbi Algorithm, where

- **X**: Vector to be decoded, of bipolar, or logic type, must be synchronous with the puncture pattern (if puncPat is used).
- **TRELLIS**: Trellis structure generated with function TRELLIS.
- **tbLen**: Trace back depth (in symbol number) when decoding, the decoded will be delayed by tbLen*K bits when MODE is 'cont'. K equals to log2(TRELLIS.numInputSymbols). Typical value of trace back length is 5~10 times of constraint length.
- **MODE**: 'cont', 'term', 'trunc' or 'tailbit'. All modes except 'tailbit' assume the decoding state starts from state 0.

  'cont' is used for continuously invoking of the function, the decoding delay is tbLen*K.

  'term' is used when there are at least max(constraint length - 1)*K zeros tail bits in the uncoded bits, decoding delay is removed.

  'trunc' estimate the last tbLen*K decoded bits from the input trace with the best metric, decoding delay is removed.

  'tailbit' is used for tail biting encoding, decoding delay is removed.

- **inType**: Data type of X, 'bipolar' or 'logic'.

  'bipolar' indicates that X consists of real type data, positive represents logic 0, negative represents logic 1, data in X should be within[-1,1].

  'logic' indicates that X consists of 1's and 0's. For quantified non-negative data(such as, 0~2^Nbits-1, 0 represents the most confident logic 0, 2^Nbits-1 represents the most confident logic 1), set MODE with 'logic' and use X/(2^Nbits-1) instead of X.

- **puncPat**: Puncture pattern vector, must be the same as that when encoding, set [] when puncture is not used and initState is used. Generally, the length of puncPat is a multiple of N, where N equals to log2(TRELLIS.numOutputSymbols).

[Y, finalState] = VITDEC(X,TRELLIS,tbLen,inType,'cont',puncPat,initState) is used for consecutive long input data. Each invoking of this function, set initState with finalState obtained from the preceding run. initState is a two element structure consists of the final input trace and state metric.

For example, if X=[X1 X2 X3], t=TRELLIS,

[Y, finalState] = vitdec(X,t,tbLen,inType,'cont',puncPat)

[Y1,finalState1] = vitdec(X1,t,tbLen,inType,'cont',puncPat)

[Y2,finalState2] = vitdec(X2,t,tbLen,inType,'cont',puncPat,finalState1)

[Y3,finalState3] = vitdec(X3,t,tbLen,inType,'cont',puncPat,finalState2)

then Y=[Y1 Y2 Y3] and finalState=finalState3.

> **Note**
> For consecutive processing, do make sure the length of input data (each piece of total input) is a multiple of the number of 1's in punctPat, i.e. sum(puncPat), and the length of puncPat is a multiple of N. With this assumption, the length of de-punctured data shall be a multiple of N=log2(trellis.numOutputSymbols). If above condition is not satisfied, consecutive decoding may fail. For 'tailbit' decoding, similar condition must be satisfied.

Examples
A coding/decoding process for the *(2,1,3)* (users) code

```
ConsLen = [4]; % 3 registers
CodeGen = [13,17]; % octal [1 0 1 1] and [1 1 1 1];
trellis = poly2trellis(ConsLen, CodeGen);
uBits = [1 0 0 1 1 0 1 0 1 1 0 0 0 1 1 0 1 0 0 0]; % 3 tail 0's for return coder state to zero,
length is 20
cBits = convenc(uBits,trellis); % encoding, output length is 40
errPattern = [0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]; %
3 errors
cBits_error = mod(cBits + errPattern,2);
tbLen = 10; % number of symbols to trace back
Mode = 'term'; % the infomation bits are terminated with ConsLen symbols
inType = 'logic'; % input are logic 0's and 1's
puncPat = []; % no puncture in encoding
dBits = vitdec(cBits_error,trellis,tbLen,Mode,inType,puncPat); % decoding
decode_error = dBits - uBits; % verify decoding results
```

results

```
decode_error =
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Compatibility

See also
convenc (users), *poly2trellis* (users)

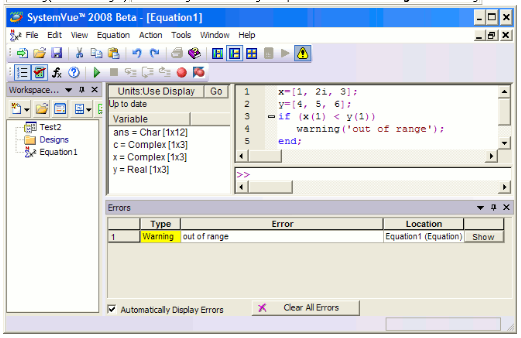
warning

Syntax
error('message')

Definition
Posts the warning message to the error log and also places the yellow warning symbol on the menu button.

Examples:

Formula	Result
warning('out of range')	the message "out of range" is posted to the **Error Log** as a warning



Compatibility
Strings

See Also
error (users)

wgn

Generates white Gaussian noise

Syntax

Y = WGN(M,N,PWR)

Y = WGN(M,N,PWR,IMP)

Y = WGN(M,N,PWR,IMP,STATE)

Y = WGN(..., POWERTYPE)

Y = WGN(..., OUTPUTTYPE)

Definition

Y = WGN(M,N,PWR) generates an M-by-N matrix of white Gaussian noise. PWR specifies the output power in decibels relative to a watt. The default load impedance is 1 ohm.

Y = WGN(M,N,PWR,IMP) is the same as the previous syntax with impdedance specified.

Y = WGN(..., POWERTYPE) is the same as the previous syntaxes with powertype specified. Choices for powertype are 'dBW', 'dBm', and 'linear'.

Y = WGN(...,OUTPUTTYPE)) is the same as the previous syntaxes with outputtype specified. Choices for outputtype are 'real' and 'complex'.

Examples

Compatibility

See also
randn (users), *awgn* (users)

xcorr

Compute cross-correlation

Syntax

c = xcorr(x, y, maxlags, 'option')

[c, lags] = xcorr(...)

Definition

1. **xcorr** estimates the cross-correlation sequence of a random process. Autocorrelation is a special case of cross-correlation.
2. **y**, **maxlags**, and **'option'** are optional parameters.
3. When only **x** is specified i.e. c = xcorr(x) then **c** is the autocorrelation sequence for the vector **x**.
4. The various **'options'** are:
   1. **'biased'** - Biased estimate of the cross-correlation function Rxy_biased(m) = [1 / N] * Rxy(m)
   2. **'unbiased'** - Unbiased estimate of the cross-correlation function Rxy_unbiased(m) = [1 / (N - | m |)] * Rxy(m)
   3. **'coeff'** - Normalizes the sequence so the autocorrelations at zero lag are identically 1.0.
   4. **'none'** - Use the raw unscaled cross-correlations. This is the default.
5. **maxlags** - Limits the autocorrelation lag range to [-maxlags:maxlags].
6. [c, lags] = xcorr(...) returns two variables **c** and **lags**. **lags** is a vector of the lag indices at which c was estimated. The ' ... ' represent the x, y, maxlags, 'option' arguments.

Examples:

Formula	Result
x = [1, 2i, 3] y = [4, 5, 6] c = xcorr(x, y)	c = [6 + i333.1e-18, 5 + i12, 22 + i10, 15 + i8, 12 - i333.1e-18]

See also
conv (users)

xor

Syntax
y = xor(A, B)

Definition
This function performs an exclusive OR operation on arrays A and B.
It returns a vector of logical values that are true if only one of the corresponding values in A OR B is nonzero, but not both. Otherwise, the value is false. A and B have to be vectors or arrays of the same size.

Examples:

```
A = [0 0 pi eps], B=[0 -2.4, 0, 1]
C = xor(A, B) = [0, 1, 1, 0]
```

zp2ss

Convert zero-pole-gain filter parameters to state-space form

Syntax

[a, b, c, d] = zp2ss(z, p, k)

Definition

Examples

Compatibility

See also
ss2zp (users), *tf2ss* (users), *zp2tf* (users)

zp2tf

Convert zero-pole-gain filter parameters to transfer function form

Syntax

[num, den] = zp2tf(z, p, k)

Definition

Examples

Compatibility

See also
ss2tf (users), *tf2zp* (users), *zp2ss* (users)

Basic

Function Name	Description
abs (users)	absolute value or magnitude
acos (users)	inverse cosine, in radians
acosd (users)	inverse cosine, in degrees
acosh (users)	inverse hyperbolic cosine
acot (users)	inverse cotangent
acotd (users)	inverse cotangent, in degrees
acoth (users)	inverse hyperbolic cotangent
acsc (users)	inverse cosecant
acscd (users)	inverse cosecant, in degrees
acsch (users)	inverse hyperbolic cosecant
all (users)	true if all parts in a vector are nonzero
angle (users)	phase of a complex number, in radians
any (users)	true if any part in a vector is nonzero
asec (users)	inverse secant, in radians
asecd (users)	inverse secant, in degrees
asech (users)	inverse hyperbolic secant
asin (users)	inverse sine, in radians
asind (users)	inverse sine, in degrees
asinh (users)	inverse hyperbolic sine
atan (users)	inverse tangent, in radians
atan2 (users)	4-quadrant inverse tangent, in radians
atand (users)	inverse tangent, in degrees
atanh (users)	inverse hyperbolic tangent
ceil (users)	smallest integer greater than or equal to argument
class (users)	data-type (class name) of argument
conj (users)	complex conjugate
conv (users)	linear convolution (or polynomial multiplication)
cos (users)	cosine of a radian-valued argument
cosd (users)	cosine of a degree-valued argument
cosh (users)	hyperbolic cosine
cot (users)	cotangent of a radian-valued argument
cotd (users)	cotangent of a degree-valued argument
coth (users)	hyperbolic cotangent

csc (users)	cosecant of a radian-valued argument	
cscd (users)	cosecant of a degree-valued argument	
csch (users)	hyperbolic cosecant	
dbg_print (users)	output to equation debug window	
dbg_showvar (users)	output contents of a variable to equation debug window	
deconv (users)	deconvolution (or polynomial division)	
dec2hex (users)	decimal to hexadecimal conversion	
diag (users)	create diagonal matrix or extract diagonal of a matrix	
diff (users)	difference (or approximate derivative)	
eig (users)	eigenvalues and eigenvectors of a matrix	
erf (users)	error function	
erfc (users)	complementary error function	
error (users)	posts to error log or output error to command window	
exist (users)	check the existance of a variable or a builtin function	
exp (users)	exponential	
eye (users)	construct identity matrix	
eyediag (users)	build an eye diagram from time data	
false (users)	logical false	
fclose (users)	close a file or stream	
fft (users)	Discrete Fourier Transform (DFT)	
fgets (users)	read a line from a file, keep newline	
find (users)	indices of nonzero parts	
findstr (users)	find a string within another string	
fix (users)	round toward zero	
floor (users)	largest integer less than or equal to argument	
fopen (users)	open file or stream	
fread (users)	read binary data from a file or stream	
fprintf (users)	write formatted text to a file or stream	
fscanf (users)	read formatted text from a file or stream	
fwrite (users)	write binary data to a file or stream	
getindep (users)	returns the string property containing the path to the independent value of a variable x. (ie. the reference to the independent variable)	
getindepvalue (users)	returns the single independent value of a variable x.	
getunits (users)	Returns an integer corresponding to the units of a variable x. This integer may be used by setunits.	
getvariable (users)	get the value of a variable from a dataset	
hex2dec (users)	hexadecimal to decimal conversion	
hilbert (users)	compute the analytic signal from a real data vector	
histc (users)	histogram count	
ifft (users)	Inverse Discrete Fourier Transform (IDFT)	
imag (users)	imaginary part of a complex number	
inf (users)	infinity	
ischar (users)	true if argument is of type character array	
isempty (users)	true if argument is empty or array with a dimension of length 0	
isequal (users)	true if arrays contain equal values, ignoring NaNs	
isfinite (users)	true for finite parts	
isfloat (users)	true if argument is a floating point scalar or array	
isinf (users)	true for infinite parts	
isinteger (users)	true if argument is an integer scalar or array	
islogical (users)	true if argument is a logical scalar or array	
isnan (users)	true for NaN parts	
isreal (users)	true if argument is a real-valued scalar or array	fones
isscalar (users)	true if argument is a scalar	
isstr (users)	true if argument is a character array	
length (users)	length of a vector	
linspace (users)	construct linearly spaced vector	
log (users)	natural logarithm	
log2 (users)	Base-2 logarithm	
log10 (users)	Base-10 logarithm	
logspace (users)	construct logarithmically spaced vector	
lu (users)	LU matrix factorization	
max (users)	largest value of a vector	
mean (users)	arithmetic mean of a vector	
median (users)	median of a vector	
min (users)	smallest value of a vector	
mkdir (users)	make directory	
mod (users)	modulus after division	
mode (users)	mode (most frequent value) of a vector	
nan (users)	Not-a-Number	
num2str (users)	convert number to a character array	
numel (users)	total number of parts in an array	
rand (users)	uniformly distributed random numbers between 0 and 1	
randn (users)	Normally (Gaussian) distributed random numbers	
real (users)	real part of a complex number	
rem (users)	remainder after division	
reshape (users)	change dimensions of an array	
roots (users)	roots of a polynomial	
round (users)	round towards nearest integer	
runanalysis (users)	Run an analysis in the workspace tree. Useful for scripting simulations.	
sec (users)	secant of a radian-valued argument	
secd (users)	secant of a degree-valued argument	
sech (users)	hyperbolic secant	
setindep (users)	set the independent reference for a swept dependent variable to indepvar(s). A minimum of two arguments is required. This function can be used to remove all independent values of a variable by passing in a blank string for the second argument.	
setvariable (users)	write a value to a variable in a dataset	
setunits (users)	sets a variable to have units specified by unit. The unit may be an integer or a string. Integer units correspond to the units returned by the getunits function. Units do not change the underlying value of a variable, but rather, just change how the value is displayed. Example: setunits('freqaxis', 'MHz')	
sign (users)	signum	
sin (users)	sine of a radian-valued argument	
sinc (users)	sinc function (sin(pi*x) / (pi*x))	
sind (users)	sine of a degree-valued argument	
sinh (users)	hyperbolic sine	
size (users)	dimensions of an array	
skewness (users)	skewness of a vector	
sort (users)	sort a vector in ascending or descending order	
spline (users)	cubic spline interpolation	
sqrt (users)	square root	
sscanf (users)	read formatted text from a string	
std (users)	standard deviation of a vector	
str2num	convert a string to a number	

(users)	
strcmp (users)	case-sensitive string comparison
strcmpi (users)	case-insensitive string comparison
strncmp (users)	compare first N characters of a string (case-sensitive)
strncmpi (users)	compare first N characters of a string (case-insensitive)
struct (users)	construct a structure array
sum (users)	sum of the parts of a vector
svd (users)	matrix singular value decomposition
tan (users)	tangent of a radian-valued argument
tand (users)	tangent of a degree-valued argument
tanh (users)	hyperbolic tangent
tcpip (users)	construct tcpip stream object for TCP/IP communications
toeplitz (users)	construct Toeplitz matrix
true (users)	logical true
using (users)	sets the current context in an equation block to the dataset called Dataset
var (users)	variance of a vector
warning (users)	posts a warning to error log or output warning to command window
xcorr (users)	cross correlation
xor (users)	logical exclusive-OR

Communications

alignsignals (users)	align two signals by delaying earliest signal
awgn (users)	add white Gaussian noise to signal
bi2de (users)	convert binary vectors to decimal
convdeintrlv (users)	permute data with specified shift register group
convenc (users)	convolutionally encode binary data
convintrlv (users)	permute data with specified shift register group
crcdec (users)	cyclic redundancy check decoder
crcenc (users)	cyclic redundancy check encoder
de2bi (users)	decimal numbers to binary vectors
deintrlv (users)	reorder data back with specified permutation table
depuncture (users)	restores erasures based on puncture pattern
finddelay (users)	estimate delay(s) between signals
matdeintrlv (users)	reorder data by filling matrix by columns and emptying it by rows
matintrlv (users)	reorder data by filling matrix by rows and emptying it by columns
muxdeintrlv (users)	restore ordering of data with specified shift register group
muxintrlv (users)	reorder data with specified shift register group
noisebwlv (users)	equivalent two-sided noise bandwidth of lowpass filter
oct2dec (users)	convert octal to decimal numbers
poly2trellis (users)	convert convolutional code polynomials to trellis description
puncture (users)	erase specified symbols based on puncture pattern
qfunc (users)	Q function
qfuncinv (users)	inverse Q function
randerr (users)	generate bit error patterns
randint (users)	generate uniformly distributed random integers
randsrc (users)	generate random matrix using prescribed alphabet
rcosflt (users)	filter input signal with (sqrt) raised cosine filter
rectpulse (users)	rectangular pulse shaping
rsdec (users)	reed-Solomon decoder
rsenc (users)	reed-Solomon encoder
symerr (users)	compute number of symbol errors and symbol error rate
turbodec (users)	compute number of symbol errors and symbol error rate
turboenc (users)	inverse Q function
vitdec (users)	convolutionally decodes binary stream using Viterbi algorithm
wgn (users)	generates white Gaussian noise

Signal Processing

Function Name	**Description**
bartlett (users)	Bartlett Window
bilinear (users)	parameter transformation from analog filter to digital filter
blackman (users)	Blackman Window
butter (users)	Butterworth filter designer
buttord (users)	butterworth filter order and cutoff frequency calculation
cheby1 (users)	Chebyshev type 1 filter designer
cheb1ord (users)	minimum order calculation for Chebyshev Type I filter
cheby2 (users)	Chebyshev type 2 filter designer
cheb2ord (users)	minimum order calculation for Chebyshev Type II filter
conv (users)	Convolution of u and v
downsample (users)	downsample input signal
ellip (users)	elliptic or cauer filter designer
fftfilt (users)	FFT-based FIR filtering using overlap-add method
filter (users)	one dimensional digital filtering
firls (users)	multiband least square FIR filter design
firrcos (users)	raised cosine FIR Filter design
gaussfir (users)	Gaussian FIR Pulse-Shaping Filter Design
gausswin (users)	Gaussian Window
grpdelay (users)	group delay of IIR filter
hamming (users)	Hamming Window
hann (users)	Hann Window
impz (users)	impulse response of IIR digital filter
interp (users)	resample input at a higher rate with lowpass filter
interp1 (users)	one dimensional interpolation
kaiser (users)	kaiser window
kaiserord (users)	parameters that specify a kaiser window
lp2bp (users)	transform lowpass filter to bandpass filter
lp2bs (users)	transform lowpass filter to bandstop filter
lp2hp (users)	transform lowpass filter to highpass filter
lp2lp (users)	lowpass filter with normalized frequency to desired frequency
phasedelay (users)	return phase delay vector for digital filter
rectwin (users)	Rectangular Window
resample (users)	change sampling rate by rational factor
sftrans (users)	transform of lowpass filter to other type filter
sinc (users)	sinc function (sin(pi*x) / (pi*x))
square (users)	Square wave generation
ss2tf (users)	convert state-space filter parameters to transfer function form
ss2zp (users)	convert state-space filter parameters to zero-pole-gain form
tf2ss (users)	convert transfer function filter parameters to state-space form
tf2zp (users)	convert transfer function filter parameters to zero-pole-gain form
triang (users)	coefficients of a triangular window
upfirdn (users)	Upsample by zero inserting, filtering and downsampling a signal
upsample (users)	Upsample input signal by inserting R-1 zeros between elements
zp2ss (users)	convert zero-pole-gain filter parameters to state-space form
zp2tf (users)	convert zero-pole-gain filter parameters to transfer function form

Using Math Language

Math Language, along with most of its built-in functions, was designed to be compatible with m-file script syntax.

Statements

An equation block consists of one or more statements. Multiple statements placed on the same line are separated by line breaks, commas, or semicolons. The following two equation blocks are equivalent:

X = 2
Y = 3

and

X = 2, Y = 3

If you end a statement with a semicolon, it does not generate output in the command window.

Complicated statements can span multiple lines and use control structures like while loops, for loops, and if statements.

The following statement types are supported by Mathematics Language equations: assignment, comment, if, for, while, function, or return. The format of each statement type is described below.

Assignments

An assignment statement assigns a value to a variable. The syntax of an assignment statement is as follows:

variableName = Expression

For example,

```
X = 3.6;
Y = sin(3*PI);
Z = [1 2 3];
```

are all assignments.

A variable name must start with a letter, and can contain alphanumeric characters and underscore characters. An expression can contain numerical operations involving numbers, other variables, and function calls.

Vectors and matrices can be defined inline, as the following example illustrates:

```
x = [1 2 3] % a row vector
y = [1;2;3] % a column vector
z = [1 2 3; 4 5 6] % a 2x3 matrix
```

Tune Assignments

A Tune Assignments assigns a variable a specific value while marking it as Tunable. A tunable variable can then be tuned from the Tune Window and can be used by evaluations, such as Sweeps, that operate on tunable variables.

When a Tunable variable is tuned from the Tune Window, the resultant value is then updated in the Equation block where it was originally defined.

The syntax for a Tune Assignment is as follows, where Constant represents a real-valued constant:

variableName = ?Constant

For example:

```
x = ?23   % x is tunable with initial value 23
y = ?-1.5  % y is tunable with initial value -1.5
```

Comments

A comment starts with a percent character (%) and continues for the rest of that line. The following are examples of comments:

```
X = R * cos( theta ) % Here is an in-line comment
% Here is another comment
```

In the example above, only the assignment statement is executed, while both comments are ignored.

if statement

The *if* statement is a control structure that allows one set of statements to execute if a condition is met, and optionally, another set of statements to execute if the condition is not met. Valid syntax for the if statement is:

1.
```
if _expression_
_one_or_more_statements_
elseif
_one_or_more_statements_
else
_one_or_more_statements_
end
```
2.
```
if _expression_
_one_or_more_statements_
end
```
3.
```
if _expression_, _statement,_ end
```

   If *expression* evaluates to a nonzero value, the following statement block is executed, otherwise that statement block is skipped. If an else block is specified and expression evaluates to zero (false), the else block is executed. The *expression* is generally Boolean in construction.

Example:

```
x = 3
y = 2
if x == y % Note that double equals are used for comparison
x = x + 1
y = y - 2
else
x = 0
end
```

for statement

The *for* loop statement is a control structure that allows a set of statements to repeatedly execute according to the value of the *loopVariable*. The syntax is as follows:

```
for _loopVariable_ = _startValue_ : _stepValue_ : _stopValue_
_one_or_more_statements_
end
```

The *loopVariable* is initialized to the *startValue*. When *stepValue* is explicitly mentioned, *loopVariable* increments by it until it reaches or exceeds the *stopValue*. When left unspecified, *stepValue* is assumed to be of unit magnitude. The following example clarifies this:

```
x = 0, y = 0
for i = 1 : 5 % i, the _loopVariable_ takes the values [1 2 3 4 5]
x = x + 10
y = y + 100
end
```

After execution completes in the above example, i is equal to 5, x is equal to 50, and y is equal to 500.

while statement

The *while* loop statement is a control structure that executes a set of statements repeatedly based on a condition. The loop is exited when the condition is no longer satisfied. The syntax is as follows:

```
while _expression_
_one_or_more_statements_
end
```

As long as *expression* evaluates to a nonzero number or a Boolean true, the statements execute repeatedly. When *expression* evaluates to zero (false), execution continues after *end*. The following example clarifies this:

```
x = 1, y = 15;
while ( y )
x = x * y;
y = y - 1;
end
```

After execution completes, when y reaches 0 in the above example, x equals factorial of the original value of y.

function statement

The *function* statement is used to define functions or procedures. A function takes zero or more parameters as input and returns exactly list of values as the result. All variables used within a function are local; that is, you cannot use variables defined in a function in another function or in the main equation block. However, you can use variables defined in the equation block in the function. The syntax of the function statement is as follows:

```
function <resultList> = functionName( <paramList> )
_computation_statements_
_calls_to_other_functions_
end % Note: this end is optional
```

If the function takes no parameters, the parentheses must still be present after *functionName* . If the function returns a value, you should set the values in the <resultList> block.

<paramList> and <resultList> are lists of variable names separated by commas. If the last variable in <paramList> is 'varargin', then the function can take in an unspecified number of arguments, and the remaining arguments are placed into the 'varargin' cell array which can be accessed from within the function. Similarly, if the last variable in <resultlist> is 'varargout', then the function can return an unspecified number of return values which are set in the function by assigning to the 'varargout' cell array.

Inside a function definition, you may use the variables named 'nargin' and 'nargout' which hold the number of arguments passed in to the function and the number of return values requested by the caller, respectively. These may be used for error checking or other purposes.

The following example is a function used to calculate the inductor value necessary to produce a resonance at a given resonant frequency and capacitor value:

```
function resonantInductor = ResL( resonantCapacitor, resonanceFrequency )
% inductance is in nH, capacitance is in pF, frequency is in MHz
FHz = 1e6 * resonanceFrequency;
CFarads = 1e-12 * resonantCapacitor;
Omega = 2 * pi * FHz;
LHenries = 1 / (Omega^2 * CFarads);
resonantInductor = LHenries * 1e9; % the return value
end
```

The function defined above may be called as follows:

L = ResL(50, 25.8) % computes the L value in nH resonanct with 50 pF at 25.8 MHz

You may return multiple values by listing them in the result expression, as in

```
function [Ind, Q] = ResL( C, F, R )
Ind = 1
Q = 2
end
```

this is used as [MyInd, MyQ] = Resl(a,b,c)

The following example illustrates a function that takes in a variable number of arguments and returns a variable number of results.

```
function varargout = f(varargin)
SumOfArgs = 0;
for i = 1 : nargin
SumOfArgs = SumOfArgs + varargin{i}
end
varargout{1} = SumOfArgs
if nargout > 1
varargout{2} = 2 * SumOfArgs
end
end
```

Suppose we call this function as follows:

```
[a, b, c] = f(1, 2, 3, 4)
```

a would be set to 10 (the sum of the input arguments), b would be set to 20, and c would be blank since it was not assigned to in the function.

Operators

Operators and their descriptions are listed in the table below. Examples for each operator are also listed.

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>+</td><td>Addition</td><td>a + b</td></tr>
<tr><td>-</td><td>Subtraction</td><td>a - b</td></tr>
<tr><td>*</td><td>Matrix Multiplication</td><td>a * b</td></tr>
<tr><td>.*</td><td>part-by-part Matrix Multiplication</td><td>a .* b</td></tr>
<tr><td>/</td><td>Matrix Right-Division</td><td>a / b</td></tr>
<tr><td>./</td><td>part-by-part Division</td><td>a ./ b</td></tr>
<tr><td>\</td><td>Matrix Left-Division</td><td>a \ b</td></tr>
<tr><td>.\</td><td>part-by-part Left-Division</td><td>a .\ b</td></tr>
<tr><td>^</td><td>Matrix Exponentiation</td><td>a ^ 2 (means a-squared)</td></tr>
<tr><td>.^</td><td>part-by-part Exponentiation</td><td>a .^ b</td></tr>
<tr><td>'</td><td>Matrix conjugate-transpose (hermitian)</td><td>a'</td></tr>
<tr><td>.'</td><td>Matrix transpose (no conjugation)</td><td>a.'</td></tr>
<tr><td>&</td><td>part-wise Boolean And</td><td>a & b</td></tr>
<tr><td>&&</td><td>Boolean And</td><td>a && b</td></tr>
<tr><td>|</td><td>part-wise Boolean Or</td><td>a | b</td></tr>
<tr><td>||</td><td>Boolean Or</td><td>a || b</td></tr>
<tr><td>~</td><td>Boolean Not</td><td>~a</td></tr>
<tr><td>=</td><td>Assignment Operator</td><td>a = 2</td></tr>
<tr><td>==</td><td>Boolean Comparison</td><td>a == b</td></tr>
<tr><td>></td><td>Boolean Greater Than</td><td>a > b</td></tr>
<tr><td>>=</td><td>Boolean Greater Than or Equal</td><td>a >= b</td></tr>
<tr><td><</td><td>Boolean Less Than</td><td>a < b</td></tr>
<tr><td><=</td><td>Boolean Less Than or Equal</td><td>a <= b</td></tr>
<tr><td>~=</td><td>Boolean Not Equal</td><td>a ~= b</td></tr>
</table>

Vectors, Matrices, and Multidimensional Arrays

Mathematics Language supports vectors, matrices, and multidimensional arrays. Column vectors are treated as Nx1 matrices, while row vectors are 1xN matrices. Vectors and matrices can be defined inline using bracket notation, as shown below.

```
a = [1;2;3] % a is a column vector containing the parts 1, 2, and 3
b = [2.5 3 8] % b is a row vector containing the parts 2.5, 3, and 8
c = [1, 2, 3] % c is a row vector containing 1, 2, and 3. Commas are optional
M = [1 2 3; 4 5 6; 7 8 9] % M is a 3x3 matrix with the first row containing 1, 2, and 3.
M = ['help1'; 'help2'; 'help3'] % M is a 3 by 5 character array
M = ['help1' 'help2' 'help3'] % M is the string 'help1help2help3'
```

Note that semicolon denotes the end of a row, while comma separates row parts.

Indexing into Numeric Arrays

An part in an array variable is accessed with the following syntax:

```
_matrixVariable_(index1, index2, ..., indexN)
```

where *matrixVariable* is the name of the N-dimensional array.
A colon may be used to indicate every part in the dimension. If only one indexing dimension is specified, then the array is linearly indexed, which means that the array is treated as a flat list (in column-wise order) and the N'th part of the list is returned.

> Note: If only one indexing dimension is specified and it is a colon, then the array is returned as a single column vector with N parts, where N is equal to the number of parts in the array.

The following example illustrates indexing into arrays.

```
M = [1 2 3; 4 5 6; 7 8 9] % M is a 3x3 matrix with the first row containing 1, 2, and 3.
a = M(2,1) % a equals 4
b = M(1,:) % b is the row vector [1,2,3]
c = M(:,[1;3]) % c is the 3x2 matrix formed by taking the columns of the 1st and 3rd columns of
all the rows [1,3; 4,6; 7,9]
d = M(:) % d is the column vector [1;4;7;2;5;8;3;6;9]
e = M(6) % e equals 8 because it is the 6th part when M is traversed column-by column
M(1,1) = 5 % sets the value of the part in the first row and first column of M to 5
```

Vectors may also be used for specifying multiple parts in a dimension. The following example illustrates this:

```
M = [1,2,3; 4,5,6; 7,8,9]
a = M( [1; 3], [1; 2] ) % a is the matrix [1,2; 7,8]
```

Multi-dimensional arrays are formed by combining arrays of smaller dimensions in nested fashion using [s and semi-colons. For instance, a three dimensional array of size 3x2x2 would have two levels of []s:

```
M3D = [ [1, 2, 3; 4, 5, 6]; [-1, -2, -3; -4, -5, -6] ]
```

Searching Vectors and Indexing into Sweeps

Sometimes it is useful to know at what index or indices in an array a particular value is contained. To find what indices a vector contains a certain value or range of values, the *find* function may be used. This is especially useful for indexing into sweeps to extract desired data. The following example illustrates a simple case using the *find* function:

```
F = [100; 200; 300; 400; 500; 600]
i = find(F == 300)    % i equals 3
n = find(F >= 200 & F <= 350)   % n is the vector [2, 3]
X = F(n)          % X is the vector [200;300]
```

Suppose V1 is a waveform of voltages as a function of time. V1 contains data for 101 timepoints: 0 ns through 100 ns in steps of 1 ns. That is, V1 has an independent value T, the time vector, of length 101. Therefore, V1 is a 101x1 array. The following example shows how to extract a subset of the voltage waveform and construct a new time independent variables corresponding to that subset:

```
% Suppose V1 and T already exist, as described above
time_indices = find(T >= 10e-9 & T <= 20e-9);  % indices where T is between 10 and 20 ns
V1_subset = V1(time_indices);  % extract waveform between 10 and 20 ns
time_subset = T(time_indices); % ditto for the time indep
setindep('V1_subset', 'time_subset');  % now if we plot V1_subset, we see a nice x-axis
```

We can use the same approach for indexing into multi-dimensional sweeps. The key is to use the *find* function to extract the correct indices.

Indexed Assignments

Mathematics Language supports assigning a value or values into arrays. If you assign data to parts outside the current dimensions of an array, the array is automatically re-sized to accommodate the new data, while any new parts in the array are initialized to zero.

When assigning from one array to another in the form A = B, the following rules must be obeyed:

- The number of subscripts specified for array B not including trailing 1's may not exceed the number of dimensions of B
- The number of non-scalar subscripts specified for A is equal to the number of non-scalar subscripts specified for B
- The length and order of all non-scalar subscripts specified for A is equal to the length and order of all non-scalar subscripts specified for B

The following code example illustrates various aspects of indexed assignments.
Initially the variable x does not exist.:

```
x(2,3) = 5; % x is created to be a 2x3 matrix with the entry at (2,3) equal to 5 and the other
parts equal to zero so x is [0, 0, 0; 0, 0, 5]
x(:,2) = [11; 22] % x is now [0, 11, 0; 0, 22, 5]
x(1, [1 3]) = [100 200] % x is now [100, 11, 200; 0, 22, 5]
x(1:6) = 1:6 % x is now equal to [1 3 5; 2 4 6]
x(1,1,2) = 23 % now x is a 2x3x2 array with x(1,1,2) equal to 23
```

Range Vectors

A range defines a row vector in either of the following two ways:

start:stop

start:stepsize:stop

where *start*, *stepsize,* and *stop* are expressions. If *stepsize* is left out, it is assumed to be 1. A range creates a row vector with the first part value being equal to *start* , each successive part being *stepsize* greater than the previous part, until *stop* is reached. Ranges may also be used to index into arrays and extract desired sub-arrays. The following example illustrates the use of ranges.

```
x = 1:10 % x is the row vector [1 2 3 4 5 6 7 8 9 10]
y = 1:2:10 % y is the row vector [1 3 5 7 9]
M = [1 2 3; 4 5 6; 7 8 9] % M is a 3x3 matrix with the first row containing 1, 2, and 3.
a = M(1:2, 2:3) % a is the 2x2 matrix: [2,3; 5,6]
b = M(1:2:3, :) % b is the 2x3 matrix: [1,2,3; 7,8,9]
```

Mathematical Operations on Arrays

Mathematical operations on arrays are supported. In general, any scalar operation or function may be performed on an array, and the operation will be performed on an part-by-part basis, producing a resulting array that has the same dimensions as the original array. The following example illustrates this:

```
x = [1,2; 3,4]
y = [1,1; 1,1]
z = x + y % z is the matrix [2 3; 4 5]
z = z - 1 % z is now the matrix [1 2; 3 4]
w = sin(z) % w is the matrix [sin(1), sin(2); sin(3),sin(4)]
```

Multiplication is a special-case operator. When using the multiplication operator on a matrix or vector, matrix-multiplication is assumed. To do an part-by-part multiplication, the .* operator is used. Here is an example:

```
x = [1 2; 3 4]
y = [1;1]
z = x * y %z is the vector [3; 7]
w = x .* [1 0;0 1] % w is the same matrix as x
```

To find out the dimensions of an array, use the *size* function. To find out how many parts are in an array, use the *length* function:

```
x = [1 2 3; 4 5 6]
x_dims = size(x) % x_dims is the vector [2 3]
num_parts = length(x) % num_parts is 6
```

Cell Arrays

Cell arrays are arrays that support each part having a differing data type. Each part in a cell array is called a cell. As an example, you may have a 1x3 cell array in which the first cell is a number, the second cell is a character array, and the third cell is a structure. Furthermore, parts of cell arrays may be cell arrays themselves. Cell arrays, just like numeric arrays, may have any number of dimensions. Cell array vectors and matrices may be defined inline as shown here:

```
X = { [1 2; 3 4] 'abc' 3j } % X is a 1x3 cell array containing a 2x2 real matrix, a 1x3 character
array, and a complex scalar
Y = { {1 2}; {1 2; 3 4} } % Y is a 2x1 cell array containing a 1x2 cell array and a 2x2 cell array
```

Indexing into Cell Arrays

There are two ways to index into a cell array, described here:

```
M{indices} % returns the contents of the cell at the index specified by indices
M(indices) % returns the cell or cells at the index or indices specified by indices
```

Numeric arrays contained in cell arrays may be indexed inline as well:

```
M{2,3}(6) % returns 6th part of the array contained in the cell array M at location (2,3)
M{2,3}{2}(6) % returns 6th part of the array contained in the 2nd part of the cell array located
in the cell array M at location (2,3)
```

The following example illustrates indexing into cell arrays.

```
M = { 1 'abcd' [2j 56; 3 j] {6 7} } % M is now a 1x4 cell array
a = M{1} % a equals 1
b = M{2} % b equals the 1x4 character array 'abcd'
c = M(2) % c equals a 1x1 cell array that contains 1 part: a 1x4 character array 'abcd'
d = M{3}(1,1) % d equals 2j
e = M{4} % e equals the cell array {6 7}
```

Structures

A structure is a data type with named *fields*. Each field has a name and a value. The value may be of any type, including a cell array or another structure. Structure arrays of any number of dimensions are supported. In a structure array, all structures in the array have the same field names.

Structures may be defined inline as shown here:

```
x.field1 = 23; % x is a structure with a field named "field1" with value 23
x.hello = {1 2}; % x now has another field named "hello" whose value is a 1x2 cell array
x(3).hello = 1; % x is now a 1x3 structure array with fields "field1" and "hello". The third
part's hello field has value 1.
```

You may use the *fieldnames* function to determine what field names are in a structure. *fieldnames* returns a cell array of strings.

Structures may also be built using the *struct* function.

Network Communication and Instrument Control

The Math Language includes TCP/IP communication capabilities. This enables control of instruments.

When you create an equation set to do communication you will almost always want it to be not Auto-Calc. It should only calculate when specifically requested, otherwise every time an input variable changes it will rerun. It will also run on load. Turn off auto-calc by clicking the Check-Calculator tool button when viewing the equation set.

TCP/IP communication is done via the tcpip class, which is constructed using the *tcpip* function. A simple example follows (waitfor is a wait-for-character routine, PSAip contains the IP address string of an instrument, while PSASpciPort contains the port number to use for communications):

```
% - set up the tcpip pipe to the instrument
t = tcpip(PSAip, PSASpciPort) % build tcpip object using the PSA ip address and spci port
t.Terminator = 'CR/LF'; % set Terminator field
t.InputBufferSize = 100000; % use a big buffer
% - open the port  
fopen(t)
% - set real data format
fprintf(t, 'form:data real,64')
% - swap byte order
fprintf(t, 'form:border swap')
% - read the trace
fprintf(t, 'trace? trace1') % tell it to send the first trace
a3 = waitfor(t, '#') % the # is followed by some count chars
% - get the # of count bytes
aBytecnt = fread( t, 1, 'uchar=>ushort')
tTotal = str2num( aBytecnt)
% - if valid # count bytes, read them
if tTotal > 0 && tTotal < 7 % we will never have more than 6 digits of stuff
ascCount = fread(t, tTotal, 'uchar=>ushort'); % read n count bytes
nCount = str2num( ascCount); % convert to numeric
nCount = nCount / 8; % convert to doubles at 8 bytes each
else
nCount = 0;
end
% - finally read the actual data
if nCount > 0
dInput = fread(t, nCount, 'double') % get nCount data values
setvariable('OutData', 'aOut', dInput) % save it in our dataset
end
% - close t so we rerun cleanly
fclose(t)
```

Analyzing the previous example

We start by creating a tcpip class object connected to our PSA device. PSAip=='127.0.0.1' or some valid ip address as a char array. PSAspciPort is an integer port number. Once the object is built, we set the terminator (for telnet in this case) and the input buffer size (plenty to avoid overflow).

We do fopen(t) which opens the socket connection.

Once connected you can use

fread - read nnn values from the data stream
fwrite - write nnn values to the data stream
fprintf - write a string to the data stream
fscanf - read a string from the data stream

When finished, close the socket by using fclose. If you are totally done with the socket you can use the Math Language clear function to remove the class object entirely.

MATLAB Integration

SystemVue is shipped with MATLAB® interface integrated, which gives user the choice to leverage their MATLAB codes in SystemVue to conduct co-simulation.

Supported MATLAB Version

MATLAB integration supports MATLAB 2009a and later versions.
If more than one supported MATLAB version installed, you can switch to your desired MATLAB version according to the following steps.

- In Vista or Win7, right click on a command window in the start menu, and choose "Run as administrator" (in WinXP, just start it)
- cd to the directory where the MATLAB exe is (typically C:\Program Files\MATLAB\Rxxxx\bin) that you want to use with SystemVue
- type "matlab –regserver" (no quotes)
- type "exit" in the window that opens

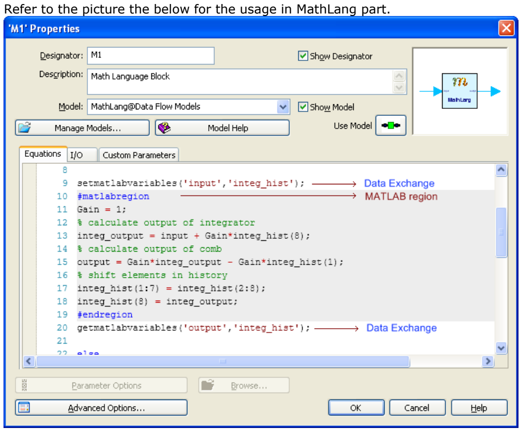
SystemVue will use that version.

Using MATLAB Integration

To use this feature in SystemVue, you only need to do

- Installed the supported MATLAB version
- Type MATLAB code within MATLAB region of equations, the region pair key words are #matlabregion and #endregion

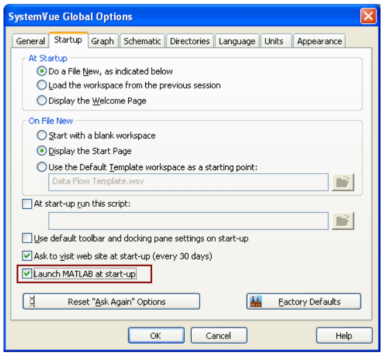
Refer to the picture the below for the usage in MathLang part.



Starting MATLAB

There are several ways to launch MATLAB.

- Launch MATLAB when SystemVue startup. You can go to menu "Tools->Options" and select startup tab.



- Starting MATLAB when SystemVue parses the MATLAB region at the first time
- Launch MATLAB by clicking on equation's context menu

Generally speaking, it will take more than 1 minute to startup MATLAB. So one waiting dialog will be shown during starting MATLAB.

End MATLAB

If MATLAB instance is created by SystemVue, it will be closed when SystemVue exits.

> Note that it will take several seconds to release MATALB interface resource.

Data Exchange between SystemVue and MATLAB

Please note that for MATLAB integration, SystemVue and MATLAB own independent variable namespace. In order to support data exchange between SystemVue and MATLAB, there are two built-in functions privided. Please refer to the two build-in functions the below for the detail.

- setmatlabvariables
- getmatlabvariables

Not all of MATLAB data type can be supported by SystemVue. Basically, the standard VARTYPE (VT_I2, VT_I4, VT_R4, VT_R8, etc) can be supported whatever the shape is scalar or array.

- Set MathLang complex variable to MATLAB

```
r = real(input);
i = imag(input);
setmatlabvariables r i;
#matlabregion
    input = complex(r, i);
    output = ifft(input, 1024);
#endregion
getmatlabvariables output;
```

- Get MATLAB complex variable to MathLang

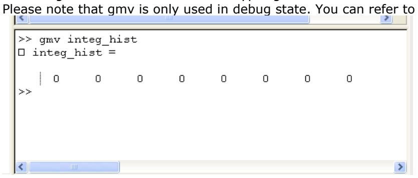
```
setmatlabvariables input;
#matlabregion
    output = fft(input, 1024);
    r = real(output);
    i = imag(output);
    a = output;
#endregion
getmatlabvariables r i;
output = r+j*i;
```

> Note that getmatlabvariable will return silently if the variable is a MATLAB instance of class

Debugging Equations with MATLAB Integration

- Fetch MATLAB variable's value.

You can go to command window and type "gmv <matlabvariablename>" to get the value. Please note that gmv is only used in debug state. You can refer to the picture the below.



Performance

Time consuming of calling MATLAB interface

In order to leverage your MATLAB code in your application, it is useful to understand the time cost to calling MATLAB interface. Generally speaking, calling the three items the below has a relatetively constant time.

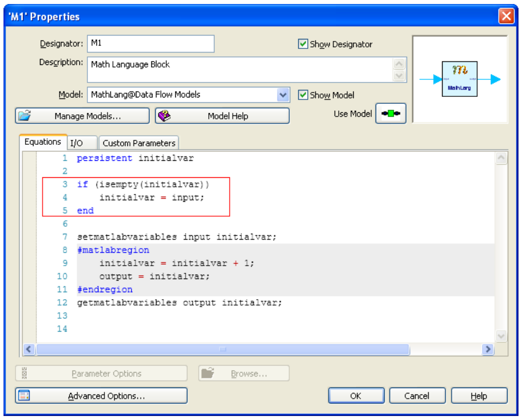
- setmatlabvariable
- getmatlabvariable
- #matlabregion - #endregion

For example, in your pc, if setmatlabvariables/getmatlabvariables takes 0.3ms and getting in/out matlabregion region requires 1ms, for the following simple application, it requires 1.6ms to complete the operation for one round. If the code is running for 1000 times, it requires 1.6s to complete it.

```
setmatlabvariables input;
#matlabregion
output = input;
#endregion
getmatlabvariables output
```

Persistent in MATLAB region

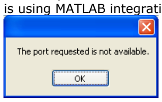
Please note that persistent variable cannot be put in MATLAB region directly. It can be used out of MATLAB region and using setmatlabvariables to transfer it to MATLAB. You can refer to the usage as the picture the below.



Initializing persistent out of MATLAB region can gain high performance.

Multi-threaded Issue for MATLAB Integration

MATLAB COM server doesn't support multi-threaed. So if more than one SystemVue instance that is using MATLAB interface, SystemVue will throw an error dialog to show that the port is not available. To avoid this error, make sure only one SystemVue program is using MATLAB integration.

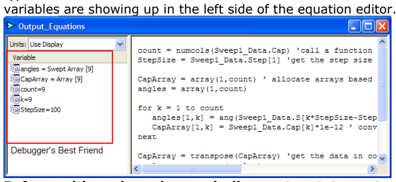


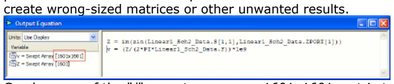
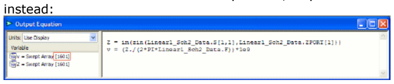
See Also

getmatlabvariables (users), *setmatlabvariables* (users)

Tips for Effective Equation Writing

As a program becomes more complex, it becomes necessary to carefully debug and test the results. Breakpoints and Debug-Print functions can be very helpful, as has already been discussed. In general, however, there are several things one should get accustomed to doing when writing equations. Below are some tips to follow when an equation is causing difficulty:

1. **Make sure the input and output equations are in separate blocks**
   It is a bad idea to have something like:
   c = ?4 ' value of some capacitor in the schematic
   s21 = Linear1_Data.S[2,1] ' s21 from analyzing the schematic
   The "c" is an input to a schematic; it MUST exist before Linear1_Data is ever created, so this equation block will not compile reliably. Any equation statements that call variables from analysis datasets should be in a separate block.
2. **Let each line compile cleanly before typing more text**
   Avoid the temptation to write a long set of statements before verifying that it works; type one line at a time and check that there are no error messages, and that the variables are showing up in the left side of the equation editor.
   

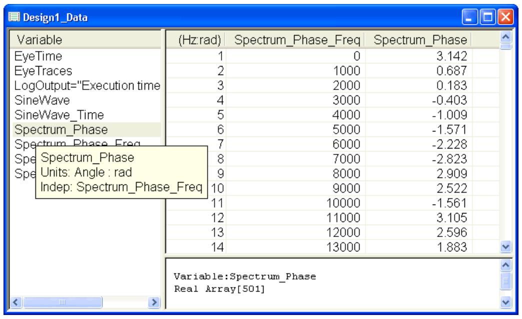
3. **Before writing a large loop or in-line vector statement, check the boundary values**
   Instead of writing a large loop then wondering why there are out of bounds errors or wrong calculations, first type something like:
   testA = myVector[firstIndex]
   testB = myVector[lastIndex]
   The values will display in the Variable view; this way you first verify that the initial and final values are as expected; then you can let the loop or vector operation run with more confidence.
4. **Don't try to pack everything into one line of code**
   It is very difficult to find the problem when there are too many calculations packed into a one line statement. By breaking up a line into several variables and lines you give yourself the chance to debug and find problems, rather than just look at a huge line that doesn't work as intended.
5. **Check dimensions of variables carefully**
   Always pay attention to the size and dimension of variables being used; a common pitfall is to use incorrect multiplication or division of vectors and thus accidentally create wrong-sized matrices or other unwanted results.
   
   Careless use of the "/" operator causes a 1601x1601 matrix to be created; the variables view alerts the user of the problem, so part-wise division can be used instead:
   

   Note that the functions **numcols(myMatrix)** and **numrows(myMatrix)** can be used to find the dimensions of a variable. For matrix operations, the number of columns of a left-hand operator should equal the number of rows of a right-hand operator, while for part-wise operations the dimensions should be identical.
6. **Use the Command Window to output or change variable values**
   See Equations User Interface for more information about the Command Window.
7. **Use the online help**
   The online help for equations is extensive. You can select a keyword in the equation editor and press F1 for context help on that keyword. General equation help is in the User's Guide manual Using Equations section.

Examining Datasets

Datasets are containers which hold data, such as the results of a simulation or a table of input. The results are stored in Variables which can be viewed in tabular form within the dataset, plotted on a graph, displayed in an output Table, etc. Examine a dataset by opening it with a double-click. You can also add new variables to a dataset (for sweeping or just for analizing the data in greater detail).

Open the Data Flow Template (via the Start Page). Double-click Design1_Data on the workspace tree and then click the variable "Spectrum_Phase" on the left-side of the window, to see its values. Hovering the mouse over a variable pops up some info, which varies according to the measurement.



In the display above the left-hand pane shows all of the result variables (including Spectrum_Phase_Freq, the frequency or independent variable associated with Spectrum_Phase, the selected variable). The right-hand pane shows whatever piece of data you have selected in the left pane. The upper left-corner box in the grid is the units of measure (Hz down and radians for the values). The lower right pane (which is usually collapsed – drag the divider bar upwards to see it) displays a summary of the variable information.

Each type of analysis creates a different dataset with differing variables which are determined by the Analysis. Often, the variable is directly associated with a particular measurement, such as BER, EVM, or P2.

Each dataset contains variables, which can be matrices, vectors, or scalars. These variables are either automatically created by simulation runs or manually by the user. Note that when a dataset is created by a simulation, the data within that dataset is always in MKS. You may *display* the data in a unit of your choice, but the actual data values are MKS values.

Click Spectrum_Phase on the left to show the tabular display of values in the grid on the top-right. It shows that the frequencies analyzed were 0, 1000, 2000, ..., 500000 Hz. The single grid-cell (top left corner of the grid) which says Hz:rad shows that the units for Frequency are Hz and the angles are shown in radians. The display on the bottom-right (which is usually collapsed) shows the type and size of the clicked data.

In addition to seeing the simulation results, Datasets can have short equations to help you analyze and diagnose issues with your circuits. For details, see *Creating Variables* (users).

Contents

Creating Datasets

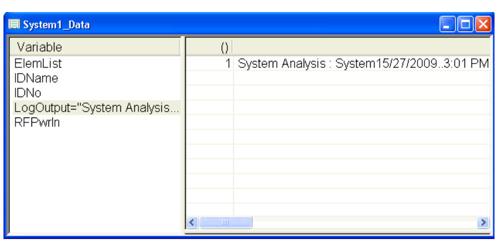
Datasets are usually created automatically when Analyses run. Some analyses (particularly SPECTRASYS) can create more than one dataset. Within the dataset are the fundamental results – measurements created by the simulation.

In addition, a blank dataset can be created manually from the workspace tree (in the docking window) via the "new item" button (although that is rarely neccesary).

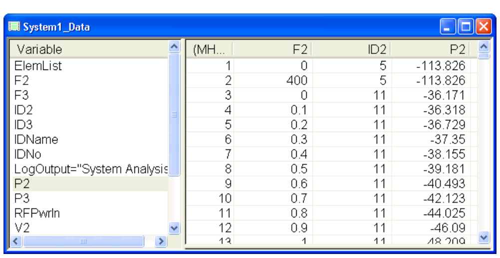
The actual data within a dataset is determined the Analyses settings. SPECTRASYS lets you limit which data is created during the simulation run. This can reduce the size of datasets significantly and also reduce their complexity.

To examine a dataset, open it by double-clicking it in the workspace tree.

Here's a minimal SPECTRASYS dataset:



If we rerun SPECTRASYS with all of the output options enabled, we get this:



Now we can't even fit the entire dataset contents in the window.

Although more complex and intimidating there are many cases where more data is better than less. However, file storage requirements go way up with this sort of data.

Creating Variables

Variable Properties Dialog Box

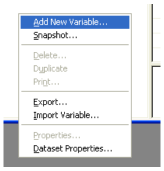
For complete description of **Variable Properties** dialog box, see *Variable Properties* (users)

Why add variables to a dataset?

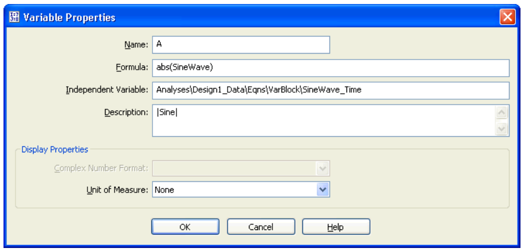
1. Add a variable to examine more closely a piece of data (such as ang(S[2,1]) to examine S21's angle). Don't forget that all measurement data is fundamentally in MKS units.
2. Add a variable to propagate it during a sweep (enable the propagate option in the sweep and it will sweep the variable along with the rest of the measurement data).
3. Add a variable to use in an optimization.

How to add a variable to a dataset

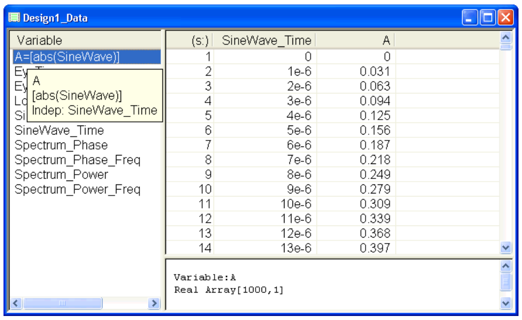
1. Open Data Flow Template / Design1_Data, as described above.
2. Right-click the white area on the left and select Add New Variable...



3. Add a variable named A.
4. Type abs(SineWave) for the formula.
5. Leave the Independent Variable field blank; it will be automatically filled in based on the indep associated with the SignWave variable.
6. Optionally, you can choose a display option for the dataset view of the variable. If the variable type is integer or floating point, select a display unit; if it's complex, select a complex number formatting option.
7. Click OK.



8. To get...



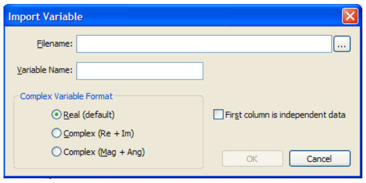
9. For most formulas, the Unit of Measure and Independent Variable will fill themselves in once the formula is parsed.

How to delete a variable from a dataset

- Right-click the variable and select Delete

Importing Variables

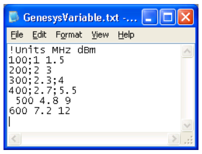
Variables can be imported to the dataset from any text file. Access this feature by right-clicking in the variable block of the data set and choosing "Import Variable".

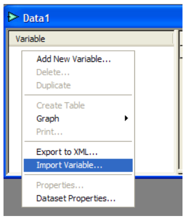


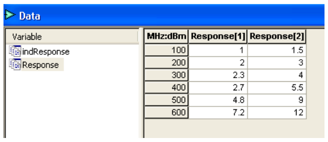
Browse and select a file. Enable "First Column is Independent Data" if the first column of the data is independent data (swept). Name the variable in the Variable Name field.

The data should be formatted as a list or matrix of numbers. Semicolons (" ; ") and spaces (" ") are used to indicate breaks between values Other characters are treated as zeroes. Begin the data with !Units *unitindep unitdep* to define a unit of measure for the data. Other rows that begin with a ! are ignored as comments.

Example (Choose Real, check First column as independent)

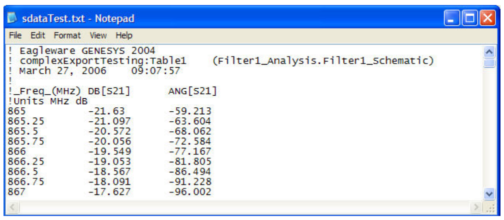


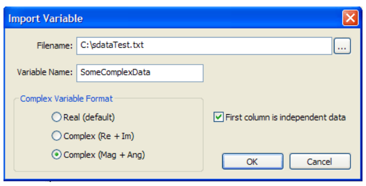


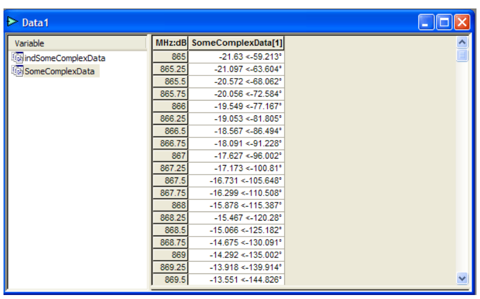


Importing Complex Variables

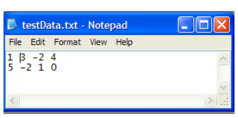
Complex data can be imported in several formats. A typical usage is shown below, where the independent vector is frequency (MHz) and the dependent is S21 in DB and ANG format. The same conventions apply here as for reals; spaces, tabs, and semicolons define breaks between entries.

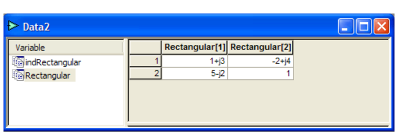






Example using rectangular coordinates (Re + Im)





Notes

1. Complex data should come in pairs of columns; two parts are needed to specify a point in the 1D complex space. A warning is given if there is an odd number of columns (excluding the independent vector).
2. To use the dB scale for complex numbers, the unit should be specified as dB; otherwise the absolute scale is used based on whatever unit is defined. For example, input impedance should have a unit of "Ohm" which can also potentially have a phase; thus it cannot be in Ohms and dB simultaneously.
3. Typical units: dB, dBm, dB10, dB20, Abs, Ohms, V, A, mil, pF, nH
4. The independent variable must be real (this will typically correspond to time or frequency, both of which are real quantities).

Using Dataset Variables

You can create variables and analyses will create variables when they run.

To graph a Dataset variable

- Right-click the variable and see *creating a graph from a dataset* (users).

To duplicate a Dataset variable

- Right-click the variable and select Duplicate

To edit a Dataset variable

- You can not edit Measurement variables (variables created during a simulation run). You can edit variables you create. Double-click the variable or right-click it and select Properties from the menu.

To delete a Dataset variable

- You should not delete Measurement variables (variables created during a simulation run). You may delete variables you create. Right-click it and select Delete from the menu.

To view a Dataset variable

- If the variable is an array, click it and the right pane will fill with the array values. If the variable is a scalar the value should be shown in the list on the left.

To export a Dataset variable

- Right-click the variable and select Export. This will export it into an XML data form.

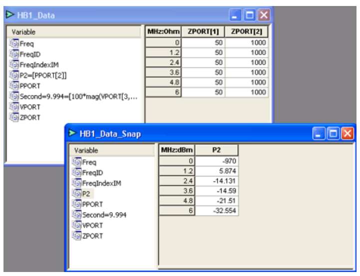
Using Datasets

Datasets are extremely useful for comparing different circuit configurations. You can run a simulation, save the data, then change some parameters, rerun the simulation and compare the two sets of data easily.

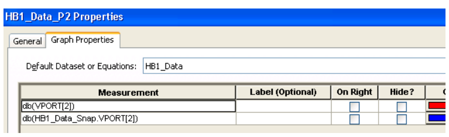
Normally, an analysis has the dataset name stored within it. You might set that name to a formula based on the parameters, but it's simpler to just Snapshot or Checkpoint the dataset.

To Checkpoint a Dataset

Right-click the dataset and select **Snapshot**. Another dataset named mydata_Snap is created. This Snap dataset contains the numerical data from the first dataset (all formulas are parsed and converted to data and the formula text is stored in the variable description).



To compare PPORT[2] for the two datasets just enter two measurements in a graph or table like this:

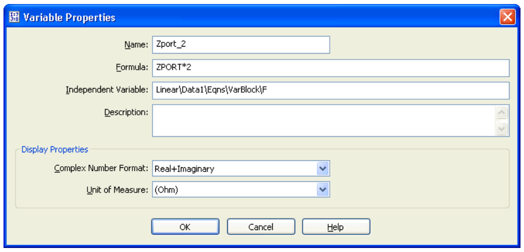


The HB1_Data_Snap.VPORT entry says to use the VPORT variable from HB1_Data_Snap.

Note we use db() here because the data in the dataset is in MKS and we want dBV for display.

Variable Properties

This window defines a variable and its display properties:



- **Name** – The variable name.
  - The name must start with a letter and contain *only* letters, numbers, and/or the underscore "_" character.
  - Names are case-sensitive. (**V1** is a different variable than **v1**.)
- **Formula** – The equation which defines the variable's value.
  - The Math Language equation may refer to other variables, functions, define vectors, matrices, etc. Please see the appropriate section in the *User's Guide* for details on using Equations.
- **Independant Variable** – One or more associated variables, which define related data, such as the X-Axis variable (which is the first indep).
  - If there is more than one independant variable, each indep should be separated by a vertical-bar character "|".
- **Description** – The discription of the variable, usage notes, etc.
- **Complext Format** – If the value is one or more complex numbers, the values can be displayed in several formats:
  - **Default** – Allows SystemVue to automatically determine the most appropriate format to use.
  - **Real + Imaginary** – Displays the values using real and imaginary values.
  - **Magnitude + Angle** – Displays the values using magnitude and angle. The magnitude is displayed using the units specified in the "Display Magnitude In" dropdown. The angle is displayed using the global Angle units, as specified in Tools / Options.
  - **Magnitude Only** – Only the magnitude is displayed.
- **Units Of Measure** – The units used for displaying the variable in the DataSet view window.
- **Display Magnitude In** – If the value is a complex number and the Complex Format includes magnitude, this specifies the units to use (for magnitude).

Graphs

Graphs display data from *datasets* (users) or *equations* (users), which are usually measurement data derived from the analysis of a design. For more information on menu items, refer *Graph Menu* (users) or *Graph Toolbar* (users) in the Appendix sections.

Contents

Annotating Graphs

The Annotation button ( ) on the Graph toolbar gives you access to the *Annotation Toolbar* (users). The Annotation toolbar provides lines, circles, and text that you can use to point out details of interest on a graph.

For example, the **Text Balloon** annotation has a "tail" which can be anchored to a data point on a graph, to the page, or not anchored by right-clicking on the balloon and selecting **Anchor Pointer** on the menu.

> To create a balloon that's initially anchored to a data point, first ensure that no marker is selected. If the trace vertices are not visible, right-click the trace and select **Show Vertex Symbols**. Right-click a trace vertex (or **WhatIF** bar) and select **Create Info Balloon**. The balloon will be anchored to the point and filled from the info box that is displayed when the mouse hovers over a data point.

> **Tip for advanced users:** To copy the text from the balloon to the **Windows** clipboard, click on the balloon, right-click on the balloon and select **Enter Text**, select the text and copy it to the clipboard using **Ctrl_V**.

Creating Graphs

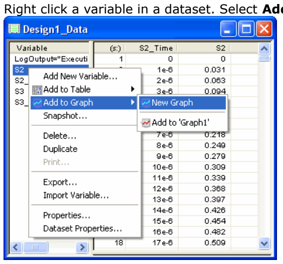
Graphs can be created manually, however the easier way to provide a context first. See sections on creating a graph from a dataset or creating a graph from a schematic.

Manually create a graph

1. Click the New Item button ( ) on the Workspace Tree toolbar.
2. Select **Add Graph...**, and the *Graph Series Wizard* (users) window will appear.
3. Select the series plot type.
4. Select the variable that you want plotted. Some plot types require more than one variable.
5. Click the OK button and the *Graph Properties* (users) window will appear.
6. If desired, change the graph **Name**, and add a title to the **Graph Heading**.
7. Click **OK**.

Create a graph from a dataset

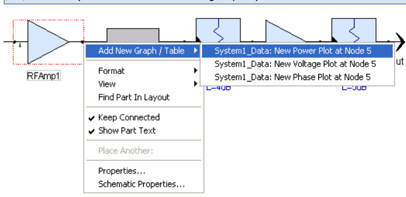
Right click a variable in a dataset. Select **Add Graph...** and click on **New Graph**.



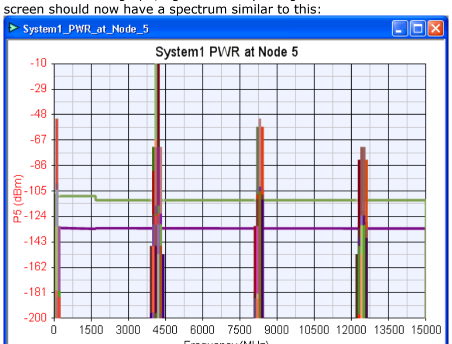
Create a graph from a schematic (RF Design Kit only)

1. Right-click a port or node on a schematic and select **Add New Graph/Table** then the measurement you want to graph from the menu.

   > The actual items available on the menu are context-sensitive, based on the part or node you clicked and the simulations available. For example, the Relevant S-Parameters option generates measurements for all S-parameter measurements that are pertinent to the indicated port. Also, the workspace must contain at least one analysis referring to this schematic design to make this feature available. (Otherwise there is nothing to plot.)



2. To create another graph, right click the port again and select a different option. Your screen should now have a spectrum similar to this:



3. Double-click a graph to change the graph's properties. Right-click a trace or legend to

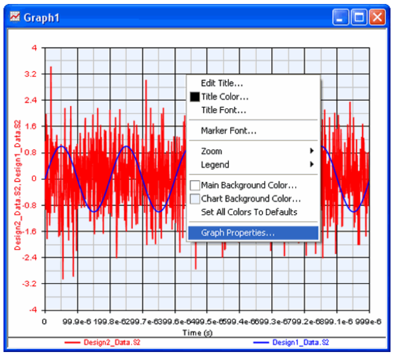
make specific changes to the appearance of the trace or legend. Hover over a symbol (a dot on the trace) to get a pop-up showing the value at that point. Check out the Graphs tutorial video for tips and techniques.

Graph Properties

Graph properties define a graph object. The **Graph Properties** window permits changes to properties such as the title or a series, i.e. a plot of a measurement variable.

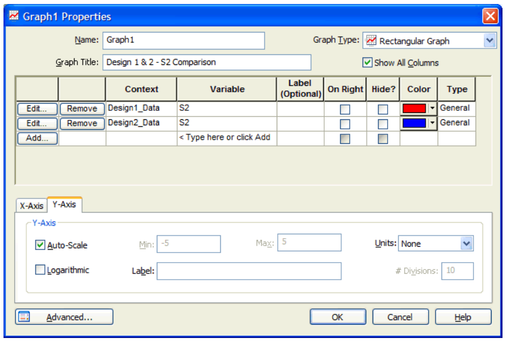
Changing Graph Properties

The Graph Properties window initially appears when a graph is created, so that you can add a series and/or customize the graph. You can make additional changes after the graph is created by right clicking the graph window or double clicking an "empty" area of the graph window and then selecting **Graph Properties...** as illustrated below.



Graph Properties Dialog

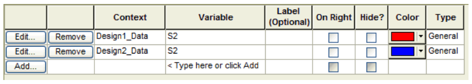
The following **Graph Properties** window was created for a *General* plot type with a *Rectangular* graph format and plots two variables from two different datasets.



- **Name** – The name of the graph object, which is shown on the workspace tree
- **Graph Title** – The plot title, which is drawn at the top of the graph (like a heading)
- **Show All Columns** – When this box is checked, infrequently-used columns in the **Series** window (such as **On Right** and **Hide?**) are shown.
- **Advanced... Button** – Clicking this button displays the **Advanced Graph Properties** dialog, as described below.

Series Settings

The following series window has defined two series for plotting.



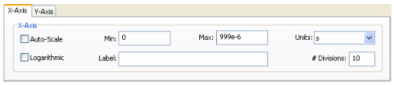
- **Edit/Add Button** – Clicking on the **Edit** or **Add** buttons will pop-up the *Graph Series Wizard* (users) for a series definition.
- **Remove Button** – Clicking on this button removes the associated series.
- **Context** – The text provides context for the **Variable** text box. If left blank, the **Variable** text box must have a fully qualified variable name. Typically, the context is the **dataset name** where the variable is defined. To graph an equation variable, set this text box to **[Equations]**. The equation hierarchy is searched for the equation variable. If the equation variable is not found, an error is logged.
- **Variable** – The text contains the name of the variable that is to be graphed.
- **Label (Optional)** – The text contains the axis label for the series. If left blank, the **Variable** text is used.
- **On Right / On Bottom** – If the box is checked, the an alternate vertical axis for the series is placed on the right side of a rectangular graph. **Polar** charts use On Bottom to indicate the use of the "lower" radial axis.
- **Hide?** – If the box is checked, the series is not plotted.
- **Color Button** – Click on this button to change the color that has been assigned to the series.
- **Type** – This informational (read only) text box states the series plot type.

Note: Checkpoint traces are NOT shown in the series grid. You can remove all the checkpoint traces on a graph by clicking the Checkpoint button on the *Graph Toolbar* (users). You can change the trace color by right-clicking a trace.

Axis Settings Tabs

The lower portion of the window contains various axis and settings tabs, which depend on the graph type.

The following is an example of a **rectangular graph** with a single vertical axis.



If both vertical axes are used, the **Y-Axis** tab name is changed to **Left Y-Axis**, and an additional tab labeled **Right Y-Axis** is added. Most of the settings are similar.

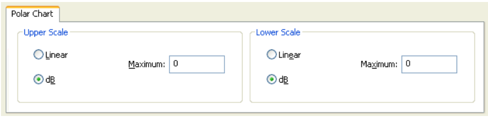
- **Auto-Scale** – When checked, the axis automatically sets its limits to match the range of the data which is being plotted
- **Label** – Use this label to customize the axis name

- **Logarithmic** – When checked, the axis is drawn with a logarithmic scale
- **Min** – Sets the lower numerical range of an axis
- **Max** – Sets the upper numerical range of an axis
- **Units** – Sets the units-of-measure used by the axis (and Min and Max)
- **# Divisions** – Sets the number of divisions to use on the axis; contains **Auto** if the divisions will be determined automatically

Tip: To control the axis tick marks, you can set the **Min** and **Max** fields to appropriate numbers, e.g. in the above examples you might want to specify the **Max** to 1000e-6 s.

Polar Tab

For a polar plot, there is simply a tab labeled **Polar**.

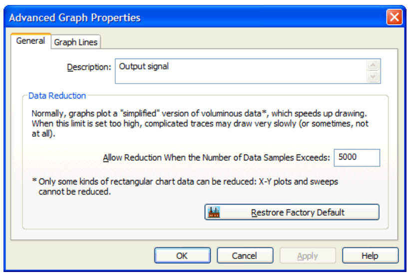


- **Upper and Lower Scale** – Polar charts have both an upper and lower scale, so that different numerical ranges may be compared on the same plot.
- **Linear or dB** – Indicates which scaling method to use
- **Maximum** – Typically 0.0 for dB and 1.0 for Linear

Advanced Graph Properties

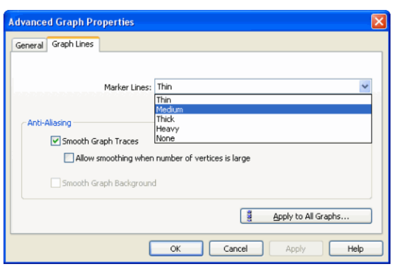
General Tab

The General tab contains generalized graph settings, such as a description field.



- **Description** – An optional description which is saved with your graph.
- **Allow Reduction When the Number of Data Samples Exceeds #** – Graphs normally plot a reduced dataset when a large number of data points must be displayed (which increases the drawing speed). Under normal circumstances, you should not be able to see a difference in the visual trace. However, markers can only be placed on a non-reduced data point. Data on a circular graph, sweeps, X-Y (trajectory or constelation) plots, or measurements without indep data cannot be reduced. The default is 5000 and the range is 5000-1000000 sample points (before data reduction is triggered).

Graph Lines Tab



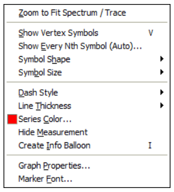
- **Marker Lines** – Sets the thickness of the graph traces: Thin, Medium, Thick, Heavy, or None
- **Smooth Graph Traces** – By default, anti-aliasing techniques are used to remove jagged pixel edges, however by default, traces with a large number of vertices are not smoothed
- **Allow smoothing when number of vertices is large** – Also smooth traces that contain a lot of points
- **Smooth Graph Background** – Smooths the "graph paper" background; only available for circular charts
- **Apply to All Graphs button** – Applies the current **Graph Lines** tab settings to all the graphs in the current workspace.

OK, Cancel, Apply and Help Buttons

Clicking the **OK** button accepts the property changes and exits the dialog. Clicking the **Cancel** button dismisses any changes and exits the dialog. Clicking the **Apply** button temporarily accepts property changes for previewing. The **Help** button links to documentation.

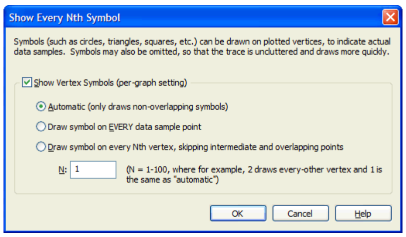
Graph Series Properties

To access the settings of a graph series (one or more data traces, based on a single measurement), right-click a series on a graph. A menu will be displayed:



- **Zoom to Fit Spectrum / Trace** – Zooms the graph, so that the selected trace fills the graph.
- **Show Vertex Symbols** – Toggles (show/hide) vertex symbols (circles, squares, triangles, etc. which indicate individual data points).
- **Show Every Nth Symbol** – Brings up a dialog box with symbol interval settings, as described below.
- **Symbol Shape** – Selects the symbol shape for the selected series.
- **Symbol Size** – Specifies the symbol size for the selected series.
- **Dash Style** – Specifies the dash style for the selected series.
- **Line Thickness** – Specifies the line thickness for the series trace(s).
- **Series Color...** – Specifies the series color.
- **Hide Measurement** – Hides the selected series. To make it visible again, double click the graph and uncheck the **Hide** checkbox. (Check **Show All Columns** if the Hide column is not visible.
- **Create Info Balloon** – Creates a Balloon annotation with info on the specified data point. Use the Annotation toolbar to change the balloon colors and other settings.
- **Graph Properties...** – Brings up graph properties (just like double-clicking the graph).
- **Marker Font...** – Brings up marker font properties.

Show Every Nth Symbol

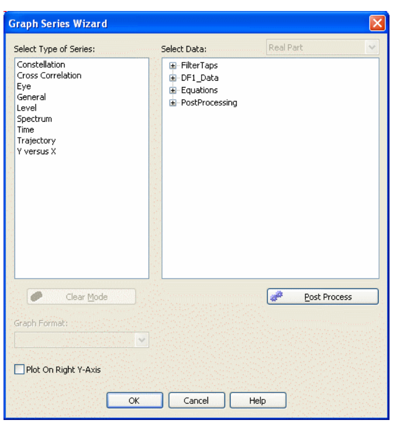


- **Show Vertex Symbols** – Same as menu toggle Vertex Symbols setting above, for convenience (since you can't see the symbol unless this is enabled).
- **Automatic** – Draws a symbol on every data point, unless it overlaps the preceeding symbol(s), in which case it is omitted. This is the default settings.
- **Draw symbol on EVERY data sample point** – Draws a symbol at every single data point (in the "reduced" data set, see note below). This can be time-consuming to draw when there are a lot of data samples.
- **Draw symbol on every Nth vertex** – Specifies that symbols should only be drawn on some of the data points, for example, 2 indicates symbols should be drawn on every-other data sample.

Note
Data reduction is used on large datasets to increase drawing speed. To adjust this setting, double-click the graph and click the **Advanced** button.

Graph Series Wizard

This wizard initializes properties for a graph object. For a new graph, the wizard is invoked by adding a graph to the work space tree or by selecting a variable from a dataset and adding a new graph. For a created graph, clicking on the **Add** or **Edit** buttons in the ***Graph Properties*** (users) dialog will pop-up the **Graph Series Wizard**. The following shows the wizard when a graph object is added to the work space tree.



Note that a complete list of series (plot) types is available and that all dataset variables are ready for selection. This wizard state can be reached by clicking the **Clear Mode** button.

A specific series can be directly chosen from the list in the **Type of Series** window, and consequently the list of available dataset variables is refined. Conversely, a dataset variable can be chosen from the **Data** window, and the list of available plot types is refined.

Wizard Components

The components are described in top-down order.

Type of Series Window

Choosing a series type limits which dataset variables can be selected for the series. It also determines how many **Data** windows are displayed (1 or 2). Note the different series types will often share some of the same variables (the measurement sets may overlap).

The list of available series types follows.

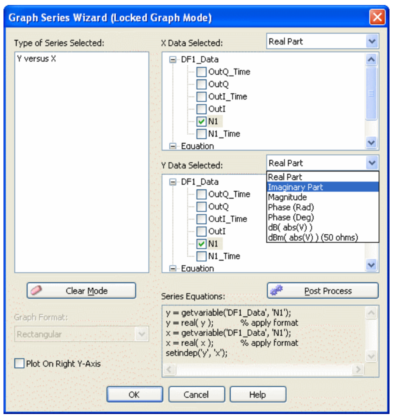
- **General**. Variables are plotted against their domains. Every variable is compatible with this type.
- **Level Diagram**. A variable generated by **SpectraSys** (RF System Analysis) which is a measurement at components along a selected path is graphed as a function of its path position.
- **Spectrum**. Plot a variable whose independent axis is Frequency. This plot type is also compatible with variables whose independent is Time and will produce a post-processed set of equations which involve taking an FFT. Various options for the FFT are available - see the Post Processed equation block.

- **Constellation**. Complex samples are plotted with the real part specifying the X-Axis coordinate and the imaginary part specifying the Y-Axis coordinate.
- **Cross Correlation**. Performs a cross correlation between two variables that are selected in separate **Data** windows.
- **Eye**. An **Eye** diagram is produced by overlaying fixed periods of a real variable.
- **Time**. Only variables with a time domain are selected for graphing.
- **Trajectory**. This is the **Constellation** plot with line segments connecting consecutive samples.
- **Y versus X**. A variable from each **Data** window is selected and graphed against each other.

Data Window

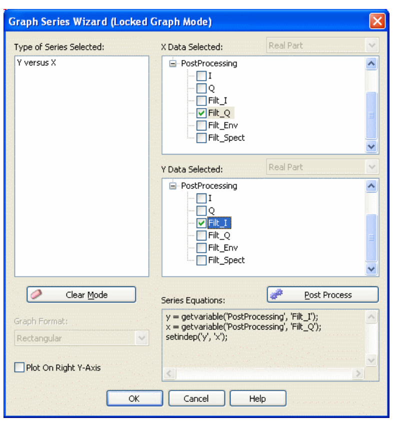
Once a series type is selected, the possible dataset variables required for the plot are displayed in one or more **Data** windows. Alternatively, if a variable is chosen then a refined list of plot types is shown in the **Type of Series** window.

Note that an un-named pull-down menu that is located at the upper right of a **Data** window can be used to modify a selected variable. This pull-down menu is activated when there is a potential need to further specify the selected variable. In the following example, only the imaginary part of the complex variable *N1* that has been selected in **Y Data** window is desired.

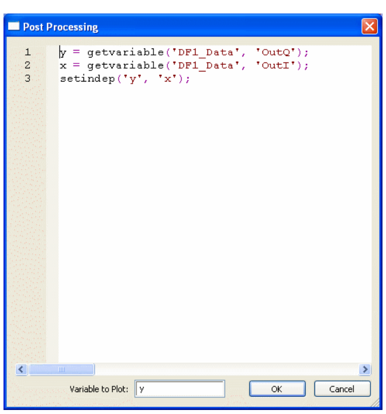


Post Process Button and Series Equations Window

The preparation for the plot may require additional calculation which is viewed in the **Series Equations** window.



While equations are automatically added, one can customize the equations. To edit the equations, click on the **Post Process** button and the following window post processing window should appear.



For the description of the equation language, see ***Using Mathematics Language*** (users). The language functions are described in ***Math Language Function Reference*** (users).

Clear Mode button

The new graph object wizard state can be reached by clicking this button.

Plot On Right Check Box

If this box is checked, the vertical axis on the right is used for this series. This check box is also available on a per series basis in the ***Graph Properties*** (users) dialog.

OK, Cancel and Help Buttons

Clicking the **OK** button proceeds to the ***Graph Properties*** (users) dialog. Clicking the **Cancel** button dismisses the wizard. The **Help** button links to documentation.

Types of Graphs

SystemVue has several types of graphs, including:

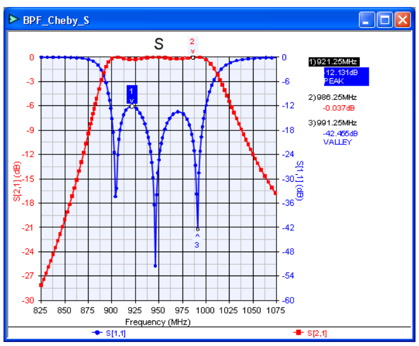
- **Rectangular Graphs** - a Cartesian coordinate plot.
- **Polar Charts** - displays complex data, such as S-Parameters or impedances.

In addition, data can be displayed in a spreadsheet-style ***Table*** (users) view. These differing output options allow you to display data in a variety of formats.

Rectangular Graphs

A rectangular graph is a Cartesian coordinate plot. You can use a rectangular graph to display two-dimensional data versus frequency (for example: magnitude or phase of a complex measurement, but not both).

In the figure below, the S-parameter insertion loss and return loss of a bandpass filter are plotted. There are 3 types of markers shown: a peak marker, a regular (fixed frequency) marker, and a valley marker. You can add to any rectangular graph. Regular markers can also be placed on circular charts.

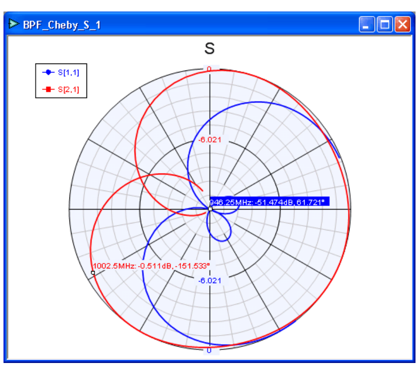


If you do not like the small circles, squares, or triangles that show each data point, hide them using the **Show Symbols On Trace** option on the Graph menu.

Polar Charts

A polar chart is used to display complex data, such as S-Parameters or impedances. In the figure below, S11 (input reflection coefficient) and S22 (output reflection coefficient) are plotted. The horizontal axis on a polar chart represents purely real numbers, while the vertical axis represents purely imaginary numbers. Numbers that lie between the two axes have both imaginary and real components.

Smith charts and polar charts generate the same plots for S-parameters (only the background and scales are changed). Additionally, certain measurements (such as Y Parameters) may be plotted on polar charts and not on Smith charts (where those measurements don't really make sense).

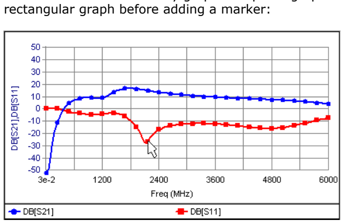


Using Markers on Graphs

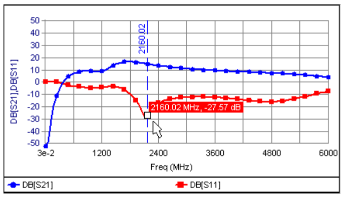
Markers are a useful way to examine and document data values on a graph.

Adding Markers to Graphs

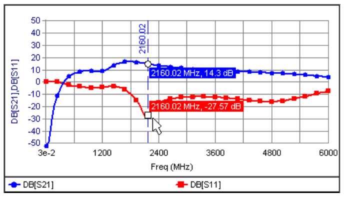
You can add markers to any graph except 3D graphs. The following figure shows a rectangular graph before adding a marker:



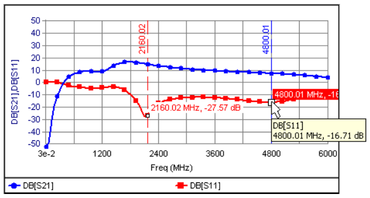
Here is the graph after placing a standard marker on the red trace:



The Mark All Traces mode displays additional marker flags on all relevant traces of chart as shown in the following figure:



Whenever a marker is selected as the currently active marker, the marker text colors are inverted (white on a colored rectangle). The figure below shows two markers. The marker on the right is selected.



To add a marker

- Click a data point on a trace. Clicking on a graph data point will create a new Marker.

To add a marker to all of the traces on a graph

- Click the **Mark All Traces** button on the graph toolbar.

To select a marker

- Click the marker you want to select.

To change the properties of a marker

- Double-click the marker to display the Marker Properties window.

To delete a marker

- Select the marker and then press the Delete key
- Alternate: click the Delete All Markers button

Marker Styles (Peak, Valley, etc.)

SystemVue has several marker types. These markers are available only for rectangular graphs. A marker's type can be changed in the Marker Properties dialog box.

- Standard - A non-moving, fixed frequency marker.
- Peak - A marker that automatically tracks the peaks of a graph, even while tuning.
- Valley - A marker that automatically tracks the valleys of a graph, even while tuning.
- Bandwidth - A composite marker for ease-of-use. Bandwidth markers are **peak** markers which drop two relative markers to measure the bandwidth of the peak. A bandwidth marker can also be a **valley** marker, simply by setting the Relative offset to be a positive number.
- Relative - A marker that automatically tracks the position of another marker and are adjusted to the relative offset (dB down). Relative markers are rarely used, except when automatically placed by SystemVue to indicate the limits of a bandwidth marker.
- Delta - Any marker style can be used as a delta marker. A delta marker displays the x / y distance to another marker.

Placing a Marker on a Trace

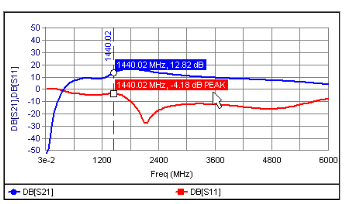
Standard marker

1. To place a Standard marker on a graph, just position the mouse over the spot where the marker is needed and click the graph trace (on or near a data point) with the left mouse button.
2. The marker can then be changed to one of the other marker types like Peak, as desired (using the Marker Toolbar or the Marker Properties window).

Peak marker

1. Place a Standard marker on a graph, as described above.
2. Click the marker and set its style to Peak using the Marker Toolbar.

The following figure is an example of what happens when a standard marker (red) is changed to a peak marker:



Notice how the marker travels to the nearest peak.

Peak/Valley detection works as follows: The "aperture" window is a box that is used for peak / valley detection. A peak or valley must be at least as large as the box, otherwise it is ignored. In general, a user should never need to adjust these, as the defaults are pretty good. A local maximum can be rejected by increasing the aperture window. Small peaks can be detected by decreasing the window size. The same criteria are used for valley detection (with a sign flip). The parameters are percentages (which are scaled by the bounds of the graph) and then used to evaluate candidate peaks / valleys and reject those that are too small to be of interest.

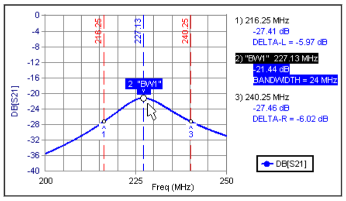
Valley marker

1. Place a Standard marker on a graph, as described above.
2. Click the marker and set its style to Valley using the *Marker Toolbar* (users).

Bandwidth marker

1. Place a Standard marker on a graph, as described above.
2. Click the marker and then click Bandwidth on the *Marker Toolbar* (users). The marker's style and name will be updated and 2 associated relative markers will be placed automatically.

Here is an example of a Bandwidth marker, along with its associated relative markers:



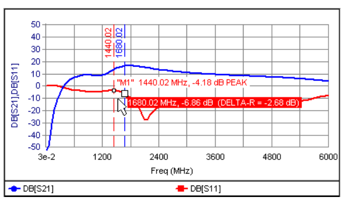
Notice that the actual measured bandwidth of 24 MHz is displayed. It is calculated directly from the positions of the two relative markers, which are both set to -6.0 dB down. You can increase the number of data points in the simulation as needed so that the relative markers are positioned with sufficient precision. You can adjust the dB down settings of both relative markers associated with the Bandwidth marker at the same time by setting the Bandwidth marker's properties. Set an individual relative marker's properties to independently set the dB down to different values.

If you need the bandwidth based on a fixed center frequency, place a bandwidth marker, change the marker type to Standard, and then type the frequency in the marker's properties window. The relative markers automatically follow the marker to its new location.

Relative marker

1. Place a Standard marker on a graph, as described above.
2. Click the marker and set its style to Relative Left or Relative Right using the Marker Toolbar. The associated relative markers will be automatically placed.

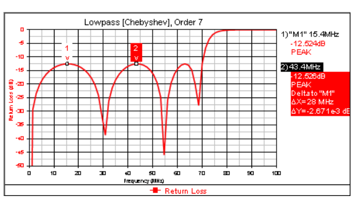
The following figure is an example of a Relative maker (on right) that is relative to the first marker ("M1"):



Notice the delta value (-2.68 dB) is displayed. That is the actual value derived from the simulation data, even though the marker's default dB down of -3.0103 dB was requested. The relative marker is always placed on an actual simulation data point.You can increase*the number of data points in the simulation as needed to get the relative marker value closer to -3 dB down.*Because this relative marker is attached to a peak marker, both markers track tuning changes in tandem.Also, notice that the original marker is automatically named M1 so the relative marker can reference it.

Delta marker

1. Double-click the existing marker that you want to measure the delta to and ensure that it has a name.
2. Place a Standard marker on a graph, as described above. This will become the "delta marker".
3. Double-click the new marker.
4. Check Show delta X (and/or delta Y).
5. Select the original marker name in the Relative To combo box.
6. Click OK.



Naming Markers

Name a marker for reference or documentation purposes. You **must** name a marker if a Relative or Delta marker references it.

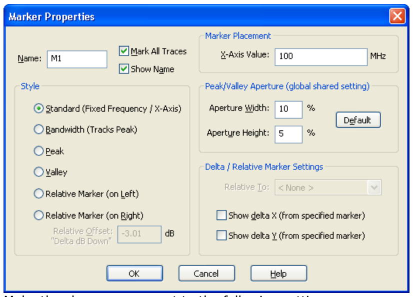
Bandwidth markers are automatically named in the format BW1, BW2, and so on. Other markers are automatically named M1, M2, and so on.You can hide marker names using the Marker Properties window; however, the name always displays on a tool tip.

Graph Marker Properties

The marker properties window lets you change the attributes of a graph marker.

To change the properties of a graph marker:

1. Double-click a marker to open its properties window.



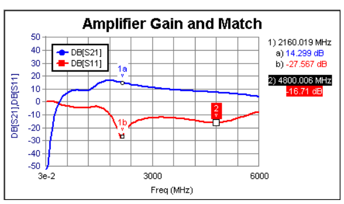
2. Make the changes you want to the following settings:
   - **Name** - The name of the marker, which is optional, unless the marker needs to be referenced by a relative or delta marker.
   - **Mark All Traces** - When checke,d the marker will mark all traces (otherwise it will only mark a single trace).
   - **Show Name** - When checked, the name of the marker will be displayed on the graph.
   - **Standard** - A normal, fixed-frequency marker.
   - **Bandwidth** - A marker which uses 2 relative markers to display the bandwidth of a peak (or valley if Relative Offset is a positive number).
   - **Peak** - A peak marker, which tracks a peak on the graph (even while tuning).
   - **Valley** - A valley marker, which tracks a valley on the graph.
   - **Relative Marker (on Left or Right)** - A tracking marker, used to measure bandwidth, etc.
   - **X-Axis Value** - The marker's location on the X-axis.
   - **Aperture Width / Height** - These values are shared between all graphs and are used to track peaks and valleys. The values are a percentage of the width / height of the graph window.
   - **Default** - Sets the Aperture Width and Height back to the Factory Default settings of 10% and 5%.
   - **Relative To** - The name of the marker to reference for relative and delta markers.
   - **Show Delta X / Y** - When checked, the distance from the "delta" marker to the reference marker will be displayed.
3. Click **OK**.

Customizing Graphs and Markers

Customize your graphs and markers to create a neater, more usable graph. There are many graph and marker options from which to choose.They include the following:

- Hiding vertical lines and trace symbols using the Graph menu.
- Placing marker text on the right using the Graph menu.
- Changing graph and marker settings using the Graph menu or Graph toolbar.
- Adding titles and annotations by right-clicking the graph and using the menu.
- Moving graph legends by dragging them into place.
- Removing symbols on traces using the Graph menu.

This following figure shows a less cluttered graph:



Zooming Graphs

You can zoom on graphs using buttons on the *Graph Toolbar* (users). Depending on which graph type you are using, some of these buttons might be grayed out.

> When zooming to a rectangle or zooming-out, both axes are zoomed, however, when zooming-in, **only** the **X-Axis** on a rectangular graph is zoomed by default. (This is because much of the time, it makes sense to keep the Y-range unchanged.)
>
> Hold down the **Ctrl** key to toggle the **Y-Axis** zoom.

> As you zoom out, the graph background may selectively skip drawing excessive details. This is intentional. Similar to using a street atlas, a state map or a world map, only the appropriate details are shown at a particular zoom setting.

Some different ways to zoom a graph follows:

1. Click one of the following buttons on the Graph toolbar or press one of the corresponding keys:

Click this button	To select this tool	Keyboard Shortcut
	Pan the graph.	P
	Zoom the graph to a rectangular region.	X
	Zoom in.	+
	Zoom out.	–

   After selecting the tool, click and drag in the graph to use the tool. When you let up on the mouse, the tool returdisappears.
   Zoom in on a rectangular area of the graph by click-dragging with the zoom tool. Click the left button to zoom in; click the right button to zoom out.
2. Click one of the following buttons on the Graph toolbar to automatically zoom to a region

Click this button	For this action	Keyboard Shortcut
	Zoom the graph to the page.	Ctrl+End
	Maximize the graph to show all the data.	Z

3. Move the mousewheel in/out to zoom the schematic in/out
4. Use the keyboard + and - keys to zoom in and out.

Graph Axis Favorites

As you work with graphs, you will find that you have sections of the graph you want to study consistently. You can define an **Axis Favorite** and easily return to it with one of the following buttons on the *Graph Toolbar* (users).

Click this button	To do	Keyboard Shortcut
	Save the current axis settings as a favorite.	F
	Use an axis favorite setting (cycle through the saved settings).	B

When you click the Save Axis Favorite button (or use a hot key found in the Graph menu), the current axis settings will be saved in a short list of favorites. If the list is full, the new settings will overwrite the oldest **Axis Favorite**.

Importing and Exporting

Contents

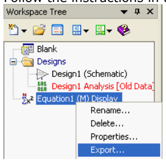
Exporting Files Using SystemVue

SystemVue can export the following file types:

- Bitmap (Active Window)
- Bitmap (Entire Screen)
- XML File

Export a file

- To export a file please follow the following steps detailed below:
  - Select the object in the Workspace Tree to be exported.
  - Click **File** on the SystemVue menu and select a file type from the **Export** menu.
  - Follow the instructions in the windows that appear.
    Or
  - Right click on the object in the Workspace Tree to be exported.
  - Select the **Export** menu.
  - Follow the instructions in the windows that appear.



Bitmap (Active Window) Export

A bitmap of the active window can be exported. To export the active window:

1. Open the window and make sure it is the active window
2. Select the File > Export > Bitmap (Active Window) menu
3. When prompted specify the directory and filename

Bitmap (Entire Screen) Export

A bitmap of the entire screen can be exported. To export the entire screen:

1. Select the File > Export > Bitmap (Entire Screen) menu
2. When prompted specify the directory and filename

XML file Export

Each SystemVue object in the workspace tree has an XML format associated with it. Workspace tree objects that can be exported to an XML format. To export an XML file:

1. Click the object in the workspace tree to be exported
2. Select the Export menu option from one of the given methods
3. Specify the name of the directory and filename of the exported object

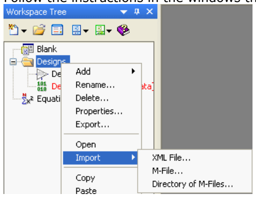
Importing Data Files Using SystemVue

SystemVue can import the following file types:

- M-File
- Directory of M-Files
- S-Data File
- XML File
- CITI File

To import a file

1. Click File on the SystemVue menu and select a file type from the Import menu.
2. Follow the instructions in the windows that appear.
   Or
3. Right click on a folder in the Workspace Tree and select the Import menu.
4. Follow the instructions in the windows that appear.



M-File Import

Imported M-Files are placed in an equation block on the workspace tree.

1. Browse to the M-file of interest
2. Click **OK**

Directory of M-Files Import

All of the M-Files located in a selected directory are imported and placed in an equation blocks on the workspace tree.

1. Browse to the M-file directory of interest
2. Click **OK**

S-Parameter Files Import

When S-Parameters are imported a dataset is create and placed in the workspace tree. This dataset is saved and loaded with the workspace and will be cached in memory to increase the simulation speed. The dataset can be deleted from the workspace. Memory cache will be used until there is a need to re-read the dataset from the workspace tree or if the dataset is not found the original file will be re-imported and cached once again.

S-Parameter can be imported in one of several ways detailed below:

Importing from a library

1. Open up the **Parts Selector** (Ctrl_Shift_A) or click the part selector button ( )
2. Change the **Current Library** to the S-Parameter library of interest
3. Click the library part of interest (the mouse cursor will change to a + sign)
4. Place the part in the schematic by clicking the schematic

> NOTE: On first use of the selected library it will be unzipped and the S-Parameter file associated with the part will be imported into the workspace tree

Or

Importing from the Main Menu

1. Select **File**, **Import**, then **S-Data File** from the main SystemVue menu
2. Browse to the S-Parameter file
   Or

Importing from a Part

1. Place a S-Parameter part in the schematic (*1-port* (rfdesign), *2-port* (rfdesign), *n-port* (rfdesign)). This can be done from the Linear Toolbar or the Part Selector
2. Double click the part to bring up the part properties
3. Click the **Browse** button to browse to the S-Parameter file

Manually Imported S-Parameters

Manually imported S-Parameters don't use a filename name. The data must exist on the workspace tree. If the dataset is deleted then there is insufficient information to correctly build the model. The model has no need of the filename and simply needs to know the name of the dataset.

1. Place a dataset part in the schematic (*NPOD* (rfdesign)). This can be done from the Linear Toolbar or the Part Selector
2. Double click the part to bring up the part properties
3. Set the **dataset** name to the name of the imported S-Parameters
4. Add an analysis and point it to the desired schematic
5. Run the analysis

Reference

NPOD

XML File Import

Each SystemVue object in the workspace tree has an XML format associated with it. Workspace tree objects that have been exported to an XML format can be re-imported into any workspace. To import an XML file:

1. Select the Import menu option from one of the given methods
2. Click XML file
3. Browse to the XML file of interest

CITI file Import

Overview

CITIfile is a standardized data format that is used for exchanging data between different computers and instruments. CITIfile stands for *Common Instrumentation Transfer and Interchange* file format.

This standard is a group effort between instrument and computer-aided design program designers. As much as possible, CITIfile meets current needs for data transfer, and it is designed to be expandable so it can meet future needs.

CITIfile defines how the data inside an ASCII package is formatted. Since it is not tied to any particular disk or transfer format, it can be used with any operating system, such as DOS or UNIX, with any disk format, such as DOS or HFS, or with any transfer mechanism, such as by disk, LAN, or GPIB.

By careful implementation of the standard, instruments and software packages using CITIfile are able to load and work with data created on another instrument or computer. It is possible, for example, for a network analyzer to directly load and display data measured on a scalar analyzer, or for a software package running on a computer to read data measured on the network analyzer.

Data Formats

There are two main types of data formats: binary and ASCII. CITIfile uses the ASCII text format. Although this format requires more space than binary format, ASCII data is a transportable, standard type of format which is supported by all operating systems. In addition, the ASCII format is accepted by most text editors. This allows files to be created, examined, and edited easily, making CITIfile easier to test and debug.

File and Operating System Formats

CITIfile is a data storage convention designed to be independent of the operating system, and therefore may be implemented by any file system. However, transfer between file systems may sometimes be necessary. You can use any software that has the ability to transfer ASCII files between systems to transfer CITIfile data.

The descriptions and examples shown here demonstrate how CITIfile may be used to store and transfer both measurement information and data. The use of a single, common format allows data to be easily moved between instruments and computers.

CITIfile Definitions

This section defines: *package* , *header* , *data array* , and *keyword* .

Package

A typical CITIfile package is divided into two parts:

- The *header* is made up of keywords and setup information.
- The *data* usually consists of one or more arrays of data.

The following example shows the basic structure of a CITIfile package:

```
Header   CITIFILE A.01.00
         NAME MEMORY
         VAR FREQ MAG 3
         DATA S RI
Data     BEGIN
         -3.54545E-2,-1.38601E-3
         0.23491E-3,-1.39883E-3
         2.00382E-3,-1.40022E-3
         END
```

When stored in a file there may be more than one CITIfile package. With the Agilent 8510 network analyzer, for example, storing a *memory all* will save all eight of the memories held in the instrument. This results in a single file that contains eight CITIfile *packages* .

Header

The header section contains information about the data that will follow. It may also include information about the setup of the instrument that measured the data. The CITIfile header shown in the first example has the minimum of information necessary; no instrument setup information was included.

Data Array

An array is numeric data that is arranged with one data part per line. A CITIfile package may contain more than one array of data. Arrays of data start after the `BEGIN` keyword, and the `END` keyword follows the last data part in an array.

A CITIfile package does not necessarily need to include data arrays. For instance, CITIfile could be used to store the current state of an instrument. In that case the keywords `VAR` , `BEGIN` , and `END` would not be required.

When accessing arrays via the DAC (DataAccessComponent), the simulator requires array parts to be listed completely and in order.

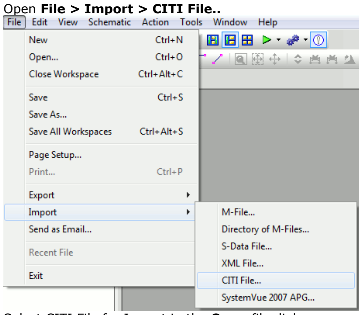
Example: S[1,1], S[1,2], S[2,1], S[2,2]

Keywords

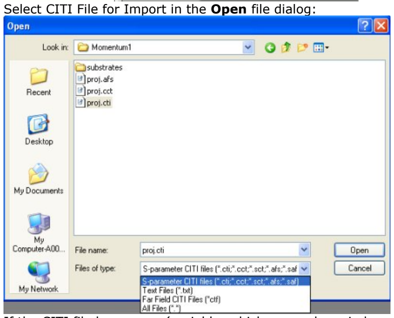
Keywords are always the first word on a new line. They are always one continuous word without embedded spaces. A listing of all the keywords used in version A.01.00 of CITIfile is shown in CITIfile Keyword Reference.

To import CITI File in SystemVue:

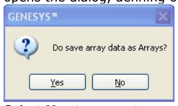
1. Open **File > Import > CITI File..**



2. Select CITI File for Import in the **Open** file dialog:

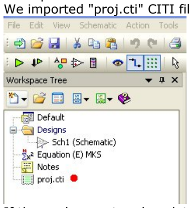


3. If the CITI file has arrays (variable, which names have indexes in square braces[]), it opens the dialog, defining output format of the array data.

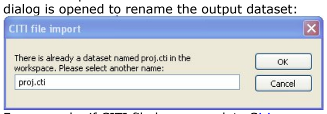


Select **Yes** to convert array data in arrays, or **No**- otherwise:

4. The imported file creates dataset with name = the file name in the SystemVue workspace tree.
We imported "proj.cti" CITI file, which created same named dataset:



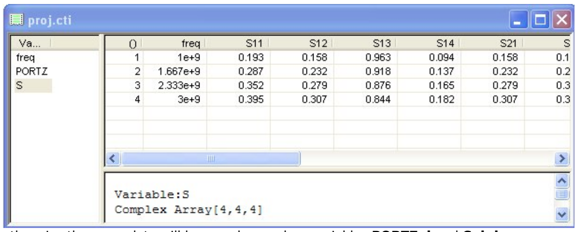
5. If the workspace tree has dataset with name of the imported CITI file, then this dialog is opened to rename the output dataset:



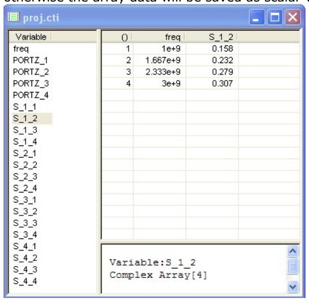
For example, if CITI file has array data Si,j

```
CITIFILE A.01.01
#Momentum: B.06.70 (*) 313.day Aug 10 2007
#Momentum Date and Time: Fri Apr 25 18:52:22 2008

NAME Momentum.SP

#  mode: RF    project: proj    reference: S_50

CONSTANT NBR_OF_PORTS 4

CONSTANT NORMALIZATION 1

VAR freq MAG 4

DATA S[1,1] RI
DATA S[1,2] RI
DATA S[1,3] RI
DATA S[1,4] RI
DATA S[2,1] RI
DATA S[2,2] RI
DATA S[2,3] RI
DATA S[2,4] RI
DATA S[3,1] RI
DATA S[3,2] RI
DATA S[3,3] RI
DATA S[3,4] RI
DATA S[4,1] RI
DATA S[4,2] RI
DATA S[4,3] RI
DATA S[4,4] RI
DATA PORTZ[1] RI
DATA PORTZ[2] RI
DATA PORTZ[3] RI
DATA PORTZ[4] RI

VAR_LIST_BEGIN
        1000000000
        1666666667
        2333333333
        3000000000
VAR_LIST_END

BEGIN
        0.080865488,  0.17510457
        0.17951702,   0.223809341
        0.270345181,  0.225034431
        0.34113765,   0.198392845
END
```

Saving its array data as Arrays will creates swept relative to independent variable **freq** vector **PORTZ** and matrix **S**:



otherwise the array data will be saved as scalar varaiables **PORTZ_i** and **S_i_j**:



CITIfile Examples

The following are examples of CITIfile packages.

Display Memory File

This example shows an Agilent 8510 display memory file. The file contains no frequency information. Some instruments do not keep frequency information for display memory data, so this information is not included in the CITIfile package.

Note that instrument-specific information (#NA = network analyzer information) is also stored in this file.

```
CITIFILE A.01.00
#NA VERSION HP8510B.05.00
NAME MEMORY
#NA REGISTER 1
VAR FREQ MAG 5
DATA S RI
BEGIN
-1.31189E-3,-1.47980E-3
-3.67867E-3,-0.67782E-3
-3.43990E-3,0.58746E-3
-2.70664E-4,-9.76175E-4
0.65892E-4,-9.61571E-4
END
```

Agilent 8510 Data File

This example shows an 8510 data file, a package created from the data register of an Agilent 8510 network analyzer. In this case, 10 points of real and imaginary data was stored, and frequency information was recorded in a segment list table.

```
CITIFILE A.01.00
#NA VERSION 8510B.05.00
NAME DATA
#NA REGISTER 1
VAR FREQ MAG 10
DATA S[1,1] RI
SEG_LIST_BEGIN
SEG 1000000000 4000000000 10
SEG_LIST_END
BEGIN
0.86303E-1,-8.98651E-1
8.97491E-1,3.06915E-1
-4.96887E-1,7.87323E-1
-5.65338E-1,-7.05291E-1
8.94287E-1,-4.25537E-1
1.77551E-1,8.96606E-1
-9.35028E-1,-1.10504E-1
3.69079E-1,-9.13787E-1
7.80120E-1,5.37841E-1
-7.78350E-1,5.72082E-1
END
```

Agilent 8510 3-Term Frequency List Cal Set File

This example shows an 8510 3-term frequency list cal set file. It shows how CITIfile may be used to store instrument setup information. In the case of an 8510 cal set, a limited instrument state is needed to return the instrument to the same state that it was in when the calibration was done.

Three arrays of error correction data are defined by using three `DATA` statements. Some instruments require these arrays be in the proper order, from `E[1]` to `E[3]` . In general, CITIfile implementations should strive to handle data arrays that are arranged in any order.

```
CITIFILE A.01.00
#NA VERSION 8510B.05.00
NAME CAL_SET
#NA REGISTER 1
VAR FREQ MAG 4
DATA E[1] RI
DATA E[2] RI
DATA E[3] RI
#NA SWEEP_TIME 9.999987E-2
#NA POWER1 1.0E1
#NA POWER2 1.0E1
#NA PARAMS 2
#NA CAL_TYPE 3
#NA POWER_SLOPE 0.0E0
#NA SLOPE_MODE 0
#NA TRIM_SWEEP 0
#NA SWEEP_MODE 4
#NA LOWPASS_FLAG -1
#NA FREQ_INFO 1
#NA SPAN 1000000000 3000000000 4
#NA DUPLICATES 0
#NA ARB_SEG 1000000000 1000000000 1
#NA ARB_SEG 2000000000 3000000000 3
VAR_LIST_BEGIN
1000000000
2000000000
2500000000
3000000000
VAR_LIST_END
BEGIN
1.12134E-3,1.73103E-3
4.23145E-3,-5.36775E-3
-0.56815E-3,5.32650E-3
```

```
-1.85942E-3,-4.07981E-3
END
BEGIN
2.03895E-2,-0.82674E-2
-4.21371E-2,-0.24871E-2
0.21038E-2,-3.06778E-2
1.20315E-2,5.99861E-2
END
BEGIN
4.45404E-1,4.31518E-1
8.34777E-1,-1.33056E-1
-7.09137E-1,5.58410E-1
4.84252E-1,-8.07098E-1
END
```

When an instrument's frequency list mode is used, as it was in this example, a list of frequencies is stored in the file after the `VAR_LIST_BEGIN` statement. The unsorted frequency list segments used by this instrument to create the `VAR_LIST_BEGIN` data are defined in the `#NA ARB_SEG` statements.

2-Port S-Parameter Data File

This example shows how a CITIfile can store 2-port S-parameter data. The independent variable name FREQ has two values located in the VAR_LIST_BEGIN section. The four DATA name definitions indicate there are four data arrays in the CITIfile package located in the BEGIN...END sections. The data must be in the correct order to ensure values are assigned to the intended ports. The order in this example results in data assigned to the ports as shown in the table that follows:

```
CITIFILE A.01.00
NAME BAF1
VAR FREQ MAG 2
DATA S[1,1] MAGANGLE
DATA S[1,2] MAGANGLE
DATA S[2,1] MAGANGLE
DATA S[2,2] MAGANGLE
VAR_LIST_BEGIN
1E9
2E9
VAR_LIST_END
BEGIN
0.1, 2
0.2, 3
END
BEGIN
0.3, 4
0.4, 5
END
BEGIN
0.5, 6
0.6, 7
END
BEGIN
0.7, 8
0.8, 9
END
```

DATA	FREQ = 1E9	FREQ = 2E9
s[1,1]	s[0.1,2]	s[0.2,3]
s[1,2]	s[0.3,4]	s[0.4,5]
s[2,1]	s[0.5,6]	s[0.6,7]
s[2,2]	s[0.7,8]	s[0.8,9]

CITIfile Keyword Reference

The following table lists keywords, definitions, and examples.

h7. CITIfile Keywords and Definitions

Keyword	Example and Explanation
CITIFILE	Example: `CITIFILE A.01.00` Identifies the file as a CITIfile and indicates the revision level of the file. The `CITIFILE` keyword and revision code must precede any other keywords. The `CITIFILE` keyword at the beginning of the package assures the device reading the file that the data that follows is in the CITIfile format.The revision number allows for future extensions of the CITIfile standard. The revision code shown here following the `CITIFILE` keyword indicates that the machine writing this file is using the A.01.00 version of CITIfile as defined here. Any future extensions of CITIfile will increment the revision code.
NAME	Example: `NAME CAL_SET` Sets the current CITIfile package name. The package name should be a single word with no embedded spaces. Some standard package names: `RAW_DATA` : Uncorrected data. `DATA`: Data that has been error corrected. When only a single data array exists, it should be named `DATA` . `CAL_SET`: Coefficients used for error correction. `CAL_KIT`: Description of the standards used. `DELAY_TABLE`: Delay coefficients for calibration.
VAR	Example: `VAR FREQ MAG 201` Defines the name of the independent variable (`FREQ`); the format of values in a `VAR_LIST_BEGIN` table (`MAG`) if used; and the number of data points (`201`).
CONSTANT	Example: `CONSTANT` *name value* Lets you record values that do not change when the independent variable changes.
#	Example: `#NA POWER1 1.0E1` Lets you define variables specific to a particular type of device. The pound sign (`#`) tells the device reading the file that the following variable is for a particular device. The device identifier shown here (`NA`) indicates that the information is for a network analyzer. This convention lets you define new devices without fear of conflict with keywords for previously defined devices. The device identifier can be any number of characters.
SEG_LIST_BEGIN	Indicates that a list of segments for the independent variable follows. Segment Format: *segment type start stop number of points* The current implementation supports only a signal segment. If you use more than one segment, use the `VAR_LIST_BEGIN` construct. CITIfile revision `A.01.00` supports only the `SEG` (linear segment) segment type.
SEG_LIST_END	Sets the end of a list of independent variable segments.
VAR_LIST_BEGIN	Indicates that a list of the values for the independent variable (declared in the `VAR` statement) follows. Only the `MAG` format is supported in revision A.01.00.
VAR_LIST_END	Sets the end of a list of values for the independent variable.
DATA	Example: `DATA S[1,1] RI` Defines the name of an array of data that will be read later in the current CITIfile *package* , and the format that the data will be in. Multiple arrays of data are supported by using standard array indexing as shown above. CITIfile revision A.01.00 supports only the `RI` (real and imaginary) format, and a maximum of two array indexes. Commonly used array names include: `S` - S parameter `E` - Error Term `Voltage` - Voltage `VOLTAGE_RATIO` - a ratio of two voltages (A/R)

CITIfile Guidelines

The following general guidelines aid in making CITIfiles universally transportable:

Line Length. The length of a line within a CITIfile package should not exceed 80 characters. This allows instruments which may have limited RAM to define a reasonable input buffer length.

Keywords. Keywords are always at the beginning of a new line. The end of a line is as defined by the file system or transfer mechanism being used.

Unrecognized Keywords. When reading a CITIfile, unrecognized keywords should be ignored. There are two reasons for this:

- Ignoring unknown keywords allows new keywords to be added, without affecting an older program or instrument that might not use the new keywords. The older instrument or program can still use the rest of the data in the CITIfile as it did

before. Ignoring unknown keywords allows "backwards compatibility" to be maintained.
- Keywords intended for other instruments or devices can be added to the same file without affecting the reading of the data.

Adding New Devices. Individual users are allowed to create their own device keywords through the # (user-defined device) mechanism. (Refer to the table immediately above for more information.) Individual users should *not* add keywords to CITIfiles without using the # notation, as this could make their files incompatible with current or future CITIfile implementations.

File Names. Some instruments or programs identify a particular type of file by characters that are added before or after the file name. Creating a file with a particular prefix or ending is not a problem. However in general an instrument or program should not require any such characters when *reading* a file. This allows any file, no matter what the filename, to be read into the instrument or computer. Requiring special filename prefixes and endings makes the exchange of data between different instruments and computers much more difficult.

A CITIfile package is as described in the main CITIfile documentation: the CITIFILE keyword, followed by a header section, usually followed by one or more arrays of data.

> **Note**
> There are some specific problems with the current version in reading and/or writing this data format. On the Agilent EEsof web site, refer to the Release Notes in Product Documentation, and to Technical Support for more information and workarounds (http://www.agilent.com/find/eesof).

Generic MDIF Format

The generic MDIF provides a generalized MDIF format for unifying the various specific MDIF formats, and overcoming some limitations of other formats. The generic format enables diverse applications to use a common data I/O interface, so long as the intent is to access/save multidimensional (multiple independent vs dependent variables) data.

The general format is as follows:

```
VAR var1Name(var1Type) = var1
ValueVAR var2Name(var2Type) = var2Value
..
VAR varNName(varNType) = varNValue
BEGIN blockName
% bVar1Name(bVar1Type) bVar2Name(bVar2Type) ....
% bVarLName(bVarLType) ...
% ...
% bVarQName(bVarQType) ... bVarPName(bVarPType)
bVar1Value bVar2Value ...
bVarLValue ..
..
bVarQValue ... bVarPValue
bVar1Value bVar2Value ...
bVarLValue ..
..
bVarQValue ... bVarPValue
...
END
```

where `var*Type` can be the token:

0 or int
1 or real
2 or string

Type bVar*Type can be one of the above as well as:

3 or complex
4 or boolean
5 or binary
6 or octal
7 or hexadecimal
8 or byte16

The variable names above constitute a name-space uniquely identified by the string blockName which is either:

- alphanumeric: all bVar*Name block variables are dependent, except bVar1Name, which is usually the most rapidly changing (innermost) independent variable.
  or
- DSCR(blockName): all bVar*Name block variables are dependent, and there is an indexing implicit independent variable.

Guidelines

- A string type variable's value must be surrounded by "".
- If there are multiple blocks, the outermost independent variables (e.g., `VAR var1Name(var1Type) = var1`) apply only to the block immediately following the variable definitions, and not to any other blocks.
- The block data (bVar*Value) lines must follow the pattern (order, number of values per line, and number of lines) of the format (%) lines. If the number of values in any data line does not match the number of dependent variables specified in the corresponding format (%) line, incorrect results will occur. A variable's value cannot be split across lines. Although there is no line length limit specified, MDIF file readers may choose to truncate at some finite length. This may result in a file read error, or, if the file was carefully crafted, truncated names and/or string-type values.
- Scale factors, which can be applied only to real numbers, may be case-insensitive suffixes as follows:

  f = 1e-15, p = 1e-12, n = 1e-9, u = 1e-6, mil = 2.54e-5, m = 1e-3,

  k = 1e3, g = 1e9, t = 1e12

  E.g.: 15mA = 15e-3, 30KHz = 30e3

  There should be no space between the number and the suffix, and extra characters are ignored. Unrecognized suffixes result in 1.0. The above is not totally consistent with the rest of ADS.
- The format of complex data is real/imag, with a column for real and a column for imaginary.
- Multidimensional data is organized by outer to inner independent variables. VAR statements go from outermost to innermost.
- Vary innermost independent variables first, proceeding toward outermost variables changing last.
- Independent variables should change monotonically.

Example

```
!=========================================================
! Example 1
REM This has 3 indepVars: v1, v2, v3(innermost) and
REM 4 depVars: dv1(integer), dv2(real), dv3(string) and
REM dv4(hexadecimal), but is read in as a string.
REM The outermost indepVars: v1, v2 apply only to the block
REM immediately following them, and not to any other block.
```

```
! There are 2 data nodes
VAR v1(0) = 1
VAR v2(1) = 2.2
BEGIN blk1
% v3(1) dv1(1) dv2(1) dv3(2) dv4(hexadecimal)
7.7 8 9.9999 "line 1" 0xabc
8.8 9 1.11 "line 2 " 0x123
END
VAR v1(0) = 2
VAR v2(1) = 3.2
BEGIN blk1
% v3(1) dv1(1) dv2(1) dv3(2) dv4(hexadecimal)
8.7 9uF 10.9999mA "line 1" 0xff
9.8 10uF 11.11mA "line 2 " 0xdef
END
!==============================================================
! Example 2
! Created Tue Mar 9 13:39:19 1999
! Data Acquired Tue Mar 9 13:38:34 1999
BEGIN NDATA_noise
% freq(real) Sopt(complex) NFmin(real) Rn(real) PortZ[1](real)
    1e+09     0.098481     0.017365       1          5    50
    2e+09      0.18794     0.068404       2         10    50
    3e+09      0.25981         0.15       3         15    50
    4e+09      0.30642      0.25712       4         20    50
    5e+09      0.32139      0.38302       5         25    50
    6e+09          0.3      0.51962       6         30    50
    7e+09      0.23941      0.65778       7         35    50
    8e+09      0.13892      0.78785       8         40    50
9.543e+09    -0.014122        0.911  9.5445     46.166    50
END
```

X-parameter GMDIF Format

This section describes:

- Choosing an X-parameter file for use with an X-Parameter part
- An overview of the X-parameter file
- Examples of various details in X-parameter files

Overview

These files contain X-parameter data for nonlinear n-port devices, or subcircuits. They are ASCII files in GMDIF format. They use extension: .xnp.
The X-parameter files completely comply by Generic MDIF Format. The specific block and variable names used in the X-parameter GMDIF files are described in this section.
This section describes Version 2.0 X-parameter GMDIF files.
An X-parameter GMDIF file can be used with an X-Parameter part to model the behavior of a nonlinear device or subcircuit using X-parameters. The file contains the X-parameters, the part is placed within the schematic.

Linking an X-parameters GMDIF File to an X-parameters Part

To link a file to the part:

1. Add X-parameters part to your schematic. It can be found in the **RF Design** library.
2. Set up the X-parameters parameters. For instructions on how to set the parameters, click Model Help in the part's dialog box.

Comments

GMDIF files support comments in two ways:

- by using "!" or
- by using "REM" statement.

The "!" can be used in the beginning of a line, or at the end of the line where as, "REM" can be used only in the beginning of a line.
Version 2.0 X-parameter GMDIF files contain a pre-defined comment section at the beginning of the files, which provides useful information about the range of operating conditions covered by the data as shown in the example below:

Example

```
! Created Fri Jul 10 15:29:17 2009
! Version = 2.0
! HB_MaxOrder = 9
! XParamMaxOrder = 3
! NumExtractedPorts = 3
! fund_1=[1e+09->1.4e+09]   NumPts=5
! VDC_3=[10->11]   NumPts=2
! ZM_2_1=50   NumPts=1
! ZP_2_1=0   NumPts=1
! AN_1_1=[3.16228e-03(-20.000000dBm)->70.7107e-03(6.989700dBm)]   NumPts=36
```

The version of the file is stated just for convenience. The statement determining the version is elsewhere. The comment "HB_MaxOrder = 9" tells you that the Harmonic Balance with MaxOrder=9 was used by X-Parameter Generator. The comment "XParamMaxOrder = 3" tells you that the X-parameter data in this file contains mixing indices up to the 3rd order.
The comment "NumExtractedPorts = 3" indicates the total number of ports used for X-parameter generation. In case of non-consecutive port numbering this value may be smaller than the highest port number.
The lower part of this comment section indicates various independent variables together with the covered sweeps for each of them. See *X-parameter Independent Variables* (users) for explanation of the variable names.

X-parameter GMDIF File Blocks

Version 2.0 of X-parameter GMDIF files contains three types of blocks:

- XParamAttributes
- XParamPortData
- XParamData

The first two blocks appear only once in the file. The third block appears as many times as the number of distinct different sweep points present in the data for all but the innermost independent variable. The following sections provide details for these blocks.

XParamAttributes Block

The **XParamAttributes** block provides the vehicle for the official statements of (1) the file version, (2) the number of ports, and (3) the number of fundamental frequencies (tones).

Example

```
BEGIN XParamAttributes
% Index(int)      Version(real)     NumPorts(int)     NumFundFreqs(int)
 0               2.0               3                 1
END
```

The sole purpose of the Index column is compliance with the Generic MDIF format.
The **NumPorts** entry indicates the highest port index in the data.

XParamPortData Block

The XParamPortData block provides reference impedances for the incident and reflected waves at each port covered by the data. The reference impedances can be complex and the power definition of the waves is used, as follows:

$$a_p = \frac{V_p + Z_p \cdot I_p}{\sqrt{8\mathrm{Re}(Z_p)}} \qquad b_p = \frac{V_p - Z_p^* \cdot I_p}{\sqrt{8\mathrm{Re}(Z_p)}}$$

In the above equations, Vp and Ip represent amplitude phasors.

Example

```
BEGIN XParamPortData
% PortNumber(int)   RefZ0(complex)              PortName(string)
 1                 50            0              "Input"
 2                 50            0              "Output"
 3                 50            0              "VDC"
END
```

The **XParamPortData** block also includes the port names. This information is particularly useful in proper hookup of the ***X-parameters part*** **(rfdesign)** in cases where more than two ports are present and a mixture of port types is used.

XParamData Block

The **XParamData** block provides the actual X-parameters. This block may appear many times in the file, each containing X-parameters at one sweep point (of all but the innermost independent variable) at a time.
Each **XParamData** block is preceded by m-1 VAR statements for m-1 independent variables, where m is the total number of independent variables. These VAR statements provide the types and the values of the independent variables. These values apply to the **XParamData** block immediately following the VAR statements, and only to that block.

Example

```
VAR fund_1(real) = 1e+09
VAR VDC_3(real) = 10
VAR ZM_2_1(real) = 50
VAR ZP_2_1(real) = 0
BEGIN XParamData
% AN_1_1(real)  FI_3(real)  FB_1_1(complex)  ...
...
...
...
END
```

The last, mth, independent variable is the innermost variable and is placed as the first variable inside the block. In the above example that variable is "AN_1_1".
The naming convention for the independent variables in X-parameter files is described in *X-parameter Independent Variables* (users).
All the dependent variables (the X-parameters) are provided inside the block. Following the mth independent variable, the names and the types of the dependent variables are specified in the header lines (lines starting with a "%" character). The header lines are specified once per block at the beginning of the block. They are then followed by as many data groups as the number of sweep points of the innermost independent variable. Each group consists of data values formatted into lines exactly in the same way as the block header lines with each entry representing a value of the correspondingly placed variable in the header lines. Complex data is specified in the rectangular format (real, imaginary) by two numbers.

Example

```
VAR fund_1(real) = 1e+09
VAR VDC_3(real) = 10
VAR ZM_2_1(real) = 50
VAR ZP_2_1(real) = 0
BEGIN XParamData
% AN_1_1(real)  FI_3(real)  FB_2_1(complex)  S_1_2_2_2(complex)
0.0657          -0.32       0.113    1.01    0.222    -0.0031
0.0667          -0.33       0.111    1.02    0.222    -0.0034
0.0677          -0.34       0.110    1.05    0.222    -0.0039
END
```

In the above example the complex number (0.111 + j1.02) is the value of the dependent variable FB_2_1 at the multidimensional point established by all the values of the independent variables, including the value of 0.0667 of AN_1_1.
The naming convention for the dependent variables in X-parameter files is described in *X-parameter Dependent Variables* (users).

X-parameter Variables

Notation

All independent and dependent variables are defined with respect to port and harmonic (or mixing) indices. For each variable these indices, separated by the underscore character "", *form a string appending the reserved name of the variable. Negative indices, if allowed, are represented by a string in which the "m" character is used in place of the minus ("-") sign, with no space between the sign and the number. For example "_m2*" represents the index "-2". For clarity of presentation the following table shows the notation used in indexing the X-parameters.

k	fundamental frequency index; 1 in the case of single tone X-parameters;all consecutive numbers must be present
p	port index - a positive integer; may not be consecutive pIn - denotes the "input" port index pOut - denotes the "output" port index
n	harmonic index; positive integer nIn - denotes the harmonic on the "input" port nOut - denotes the harmonic on the "output" port in case of multi-tone X-parameters there is a mixing index that is concatenated from harmonic indices w.r.t. to subsequent fundamentals, for example "_1_m2_2" in the three-tone case refers to the mixing product f1-2f2+2f3 - the index w.r.t. the first fundamental is expected to be non-negative and all-zero entries are not allowed.

Independent Variables

The following table lists all the supported independent variables in Version 2.0 X-parameter files. In general, all X-parameters are functions of some or all of these independent variables. Their dependence is tabulated in the X-parameter files for all sweep points of the independent variable values.
All independent variables are real numbers.

fund_k	kth fundamental frequency; assumed non-commensurate if more than one is present; fund_1 is required
VDC_p	DC voltage applied to port p; not required; mutually exclusive with *IDC_p* at the same port p
IDC_p	DC current applied to port p; not required; mutually exclusive with *VDC_p* at the same port p
AN_p_n	magnitude of a large-signal incident wave applied to port p at harmonic n; only one per each fundamental is both allowed and required; phase of this incident wave is not tabulated in the X-parameter files as this incident wave serves as a Reference Signal (Refer to **ADS document** for detailed description); power definition of incident waves is used
AM_p_n	magnitude of any other than Reference Signal large-signal incident wave applied to port p at harmonic n; required only if *AP_p_n* is used at the same port p and harmonic n; power definition of incident waves is used
AP_p_n	phase in degrees of any other than Reference Signal large-signal incident wave applied to port p at harmonic n; required only if *AM_p_n* is used at the same port p and harmonic n
GM_p_n	magnitude of the reflection coefficient of the load at port p and harmonic n; required only if *GP_p_n* is used at the same port p and harmonic n; power definition of the reflection coefficient and the reference impedance specified for port p are used; mutually exclusive with other formats of specifying load at the same port p and harmonic n
GP_p_n	phase in degrees of the reflection coefficient of the load at port p and harmonic n; required only if *GM_p_n* is used at the same port p and harmonic n; mutually exclusive with other formats of specifying load at the same port p and harmonic n
GX_p_n *GY_p_n*	alternative to *GM_p_n* and *GP_p_n*; real and imaginary parts of the reflection coefficient; mutually exclusive with other formats of specifying load at the same port p and harmonic n
ZM_p_n *ZP_p_n*	alternative to *GM_p_n* and *GP_p_n*; magnitude and phase of the load impedance; mutually exclusive with other formats of specifying load at the same port p and harmonic n
ZX_p_n *ZY_p_n*	alternative to *GM_p_n* and *GP_p_n*; real and imaginary parts of the load impedance; mutually exclusive with other formats of specifying load at the same port p and harmonic n

Dependent Variables

The following table provides the notation for the dependent variables (X-parameters) used in Version 2.0 X-parameter files. The X-parameters can be either real or complex numbers. In the latter case the rectangular format (real and imaginary parts) is used. It is not essential for any specific dependent variable to be present in an X-parameter file. In general, the default value is zero for any absent parameter that could otherwise be included in the file (some parameters are mutually exclusive with some other parameters).

FB_pOut_nOut	complex	B-type X-parameter - measured reflected wave at output port pOut and harmonic nOut as the response to all large-signal excitations (i.e., under the large-signal operating conditions); power definition of the reflected waves is used
FI_pOut	real	I-type X-parameter - DC current measured at output port pOut under the large-signal operating conditions
FV_pOut	real	V-type X-parameter - DC voltage measured at output port pOut under the large-signal operating conditions
S_pOut_nOut_pIn_nIn	complex	S-type X-parameter providing the small-signal added-contribution to the reflected wave at output port pOut and harmonic nOut due to a small-signal incident wave at input port pIn and harmonic nIn measured under the large-signal operating conditions; power definition of the incident and reflected waves is used
T_pOut_nOut_pIn_nIn	complex	T-type X-parameter providing the small-signal added-contribution to the reflected wave at output port pOut and harmonic nOut due to a phase-reversed small-signal incident wave at input port pIn and harmonic nIn measured under the large-signal operating conditions; power definition of the incident and reflected waves is used
XY_pOut_pIn_nIn	complex	Y-type X-parameter providing the small-signal contribution to the DC current at output port pOut due to a small-signal incident wave at input port pIn and harmonic nIn measured under the large-signal operating conditions; power definition of the incident waves is used; the real-valued contribution to the DC current is the real part of complex product of this X-parameter and the corresponding incident wave
Yre_pOut_pIn_nIn *Yim_pOut_pIn_nIn*	real real	alternative to XY_p_n, obsolete in Version 2.0 X-parameter files; two real numbers: the real part and negative of the imaginary part are provided instead of one complex number, as XY = Yre - j*Yim
XZ_pOut_pIn_nIn	complex	Z-type X-parameter providing the small-signal contribution to the DC voltage at output port pOut due to a small-signal incident wave at input port pIn and harmonic nIn measured under the large-signal operating conditions; power definition of the incident waves is used; the real-valued contribution to the DC voltage is the real part of complex product of this X-parameter and the corresponding incident wave
Zre_pOut_pIn_nIn *Zim_pOut_pIn_nIn*	real real	alternative to XZ_p_n, obsolete in Version 2.0 X-parameter files; two real numbers: the real part and negative of the imaginary part are provided instead of one complex number, as XZ = Zre - j*Zim

Restrictions

If the independent variable *VDC_pOut* is specified for the port *pOut* then neither the V-type (*FV_pOut*) nor the Z-type (*XZ_pOut_pIn_nIn*, *Zre_pOut_pIn_nIn*, *Zim_pOut_pIn_nIn*) X-parameters can be specified for the port *pOut*.
Similarly, if the independent variable *IDC_pOut* is specified for the port *pOut* then neither the I-type (*FI_pOut*) nor the Y-type (*XY_pOut_pIn_nIn*, *Yre_pOut_pIn_nIn*, *Yim_pOut_pIn_nIn*) X-parameters can be specified for the port *pOut*.

Instrument Scripting and Control

Overview

Many applications require to run multiple simulations sequentially. For example, in an LTE Bit Error Rate (BER) measurement over a device, one simulation can generate waveform(s) that will be downloaded into RF Signal Synthesizer(s) to modulate the RF signals that will stimulate the device. Another simulation will then use measurement equipment such as the Agilent Technologies MXA's to capture the output RF signal from the device and feed the measured data back into the simulation to be demodulated for BER analysis.

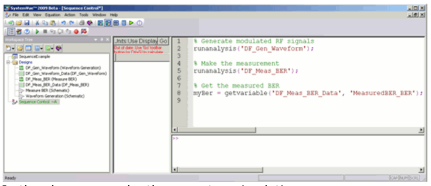
Further more, in order to characterize the device's performance, it might be necessary to adjust certain settings of some instruments several times and make the measurements after each instrument adjustment. For example, it might be necessary to change a DC bias level and see how the BER is impacted by it.

These are the applications where sequence control can be used.

SystemVue provides a powerful and flexible sequence control mechanism that is based on ***MathLang scripting* (users)**.

> **Important Note:** For all available SystemVue releases, ***MathLang scripting* (users)** can **only support LXI compliant instruments**.

A Simple Sequence



In the above example, there are two simulations:

- **DF_Gen_Waveform(Waveform Generation)** that performs a ***Data Flow Simulation* (sim)** over the **Waveform Generation(Schematic)** design
- **DF_Meas_BER(Measure BER)** that performs a ***Data Flow Simulation* (sim)** over the **Measure BER (Schematic)** design

The critical ***MathLang* (users)** built-in functions used are:

- **runanalysis** - executes the specified ***Data Flow Simulation* (sim)**
- **getvariable** - gets the simulation result data

Obviously, the BER result is stored in a variable named **MeasuredBER_BER** inside the simulation results of **DF_Meas_BER_Data(DF_Meas_BER)**.

> Notice that the **Sequence Control ~ A *MathLang Equation* (users)** page, i.e. the script, is located at the same level on the workspace tree as the workspace (i.e. project) name.

How to Run the Sequence

You can use either of the following two ways to run the sequence (the sequence ***MathLang Equation* (users)** page must be open):

- click the **GREEN** triangle button (the 4th icon) on the second tool bar
- click the **Go** button next to the Equation editor area

Example of a more Advanced Sequence

In the following sequence, we will vary the DC bias (provided by an LXI compliant DC supply), measure the BER at each of the differnt bias levels, and finally put the measured BER results into the simulation results.
The additional ***MathLang* (users)** function used in this example are:

- **setvariable** - brings the value stored in a variable into the measurement results storage area (i.e. **Data Set**) of the simulation
- **num2str** - converts a number to a string
- **fprintf** - writes a string to the opened tcpip port

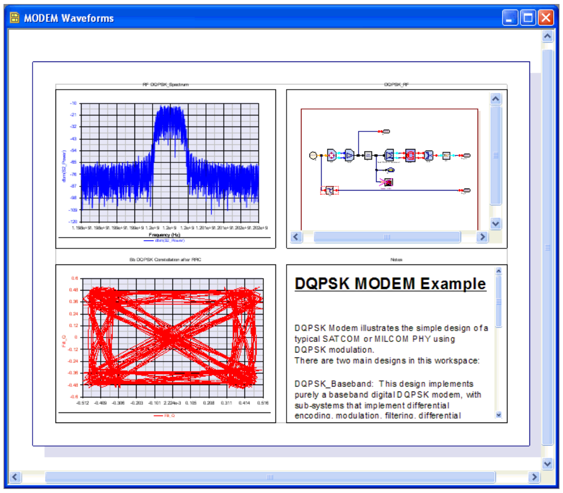
  > Note the use of the **[]** operation to concatenate the strings when creating the **dcCmdStr** command string

  > **Important Note:** Note how the accumulated BER results are stored in the **myBers** variable and how this variable is **transposed** with the **'** operator when calling **setvariable(...)** on the last line.

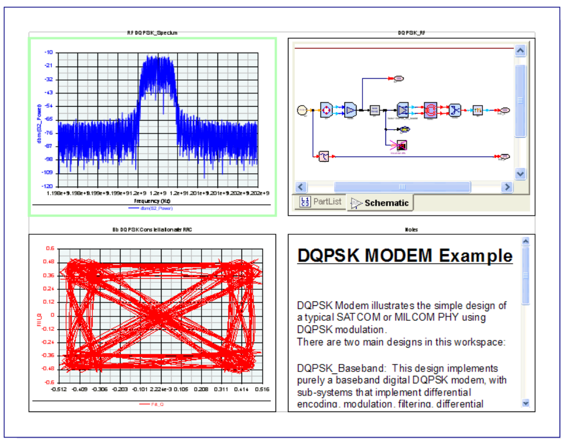
```
% 5 DC Levels starting at 3.5V at a step of 0.5V
DCLevels = (3.5:0.5:5.0);
% Number of DC's
numDCs = length(DCLevels);
% Place holder for the 5 BER's to be measured
myBers = zeros(1, numDCs);
% Generate modulated RF signals
runanalysis('DF_Gen_Waveform');
% Create tcpip communication with DC supply
dcSply = tcpip('111.222.333.444', 5025);
fopen(dcSply);
% Loop the DC levels and make the measurement
% at each level
for idx = 1:1:numDCs
dcCmdStr = [':VOLT ' num2str(DCLevels(idx))];
fprintf(dcSply, dcCmdStr);
 % make sure the DC is settled
fprintf(dcSply, '*OPC?');
statusRes = fgets(dcSply);
 % Measure BER at this DC bias
runanalysis('DF_Meas_BER');

 % Get the measured BER and store it away
myBers(idx) = getvariable('DF_Meas_BER_Data', 'MeasuredBER_BER');
end
% Now close communication with DC supply
fclose(dcSply);
% Now bring the stored 5 BER's into the simulation results
% and name the variable AllBers
setvariable( 'DF_Meas_BER_Data', 'AllBers', myBers' );
```

LiveReports

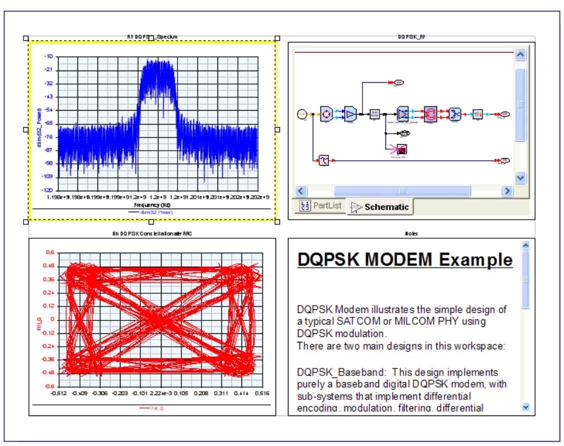
A LiveReport is a *living* page that can contain live views of various kinds of SystemVue objects. You can mix Graphs, Designs, Equations, Notes, Tables, and Datasets all in a single printable and viewable page. Below is an example of a LiveReport page from the simple Bridge-T Example.



It's a *Live* Report because you can click in any of the windows on the page and work exactly as you would work in single windows in SystemVue. The border turns green when a window is active, as seen below.



When you want to move or resize a window within a LiveReport click the black border (box) outside the window and it will turn yellow and gain handles you can drag/move.

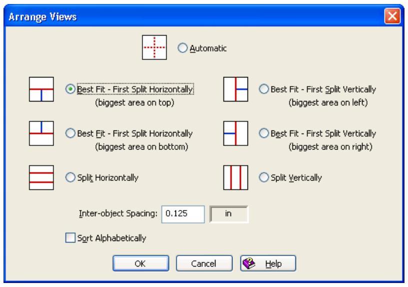


To use the LiveReport rather than one of the windows, click outside all of the windows and you will see the LiveReport toolbar and the LiveReport menu. No windows will be green or yellow.

Contents

Arranging Views

Use the LiveReport Arrange Views dialog box to arrange the sub-objects of a LiveReport.



Automatic - In general, automatic arrangement is quick, easy, and does a reasonable job of laying out the view windows.

If a window is not placed in the desired location when OK is clicked, simply drag the window into place and swap it with another, then re-do the Arrange Views.

If the LiveReport page is not divided in the desired fashion, use one of the Best Fit or Split options.

Best Fit - The best-fit options arrange sub-objects in a tiled arrangement, similar to the way Windows arranges Tiled views. The images show the order of the major and minor page splits.

Split Horizontally / Vertically - Arrange sub-objects so they are ordered in a linear fashion and are all the same size.

Inter-object Spacing - The distance (gap) between the arranged views. The units can be set in Page Properties.

Sort Alphabetically - This rarely-used option sorts the views by name, instead of the usual geometric positioning based on current pane positions. This options is mostly used to display libraries of symbols or parts.

The images to the left of the radio buttons may be double-clicked to quickly select an arrangement and close the dialog box.

Creating a LiveReport

To manually create a LiveReport:

1. Click the New Item button ( ) on the Workspace Tree toolbar, select *add* **LiveReport...**.
2. If desired, change the LiveReport name, layout, or other properties.
3. Click **OK** to create the LiveReport or click **Cancel** to not create the new LiveReport.

LiveReport Properties

There are several tab pages that you can use to change the properties of a LiveReport:

- Page Properties
- Margin Properties
- Header and Footer Properties

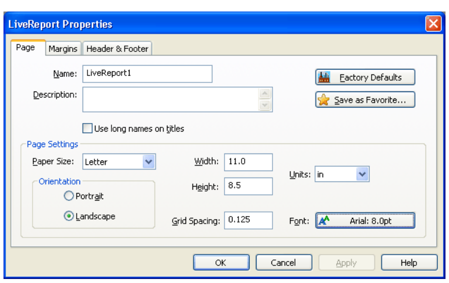
To change the properties of a LiveReport:

1. Double-click the report or click **LiveReport** on the SystemVue menu and select **Properties**.
2. Click the desired tab.
3. Make the changes you want.
4. Click **OK**.

When double-clicking the LiveReport, SystemVue uses the mouse cursor location to pick an appropriate tab. Double-click the upper or lower page area to initially display the Header & Footer page; double-click the side margins to initially display the Margins page; anywhere else displays the Page settings tab page.

Page Properties

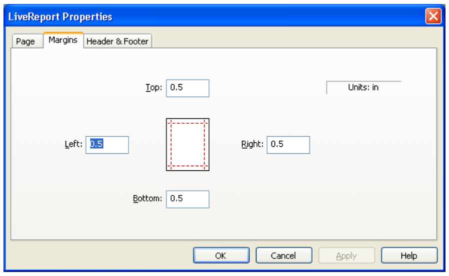
Use the LiveReport Page Properties tab page to change the general properties of a LiveReport.



1. **Name** - The name of the LiveReport.
2. **Description** - The LiveReport description (optional).
3. **Use long names on titles** - When checked, the sub-object view windows will show the full workspace pathname in their title.
4. **Paper Size** - Use this combo-box to set your page size.
5. **Orientation** - Sets the page to portrait (tall) or landscape (wide) mode.
6. **Width & Height** - The size of the paper (in current units).
7. **Grid Spacing** - The distance between grid dots.
8. **Units** - The units used by LiveReport for all of its settings, including margins and arrangement spacing.
9. **Font** - The page font, which is used for sub-object titles.

Margin Properties

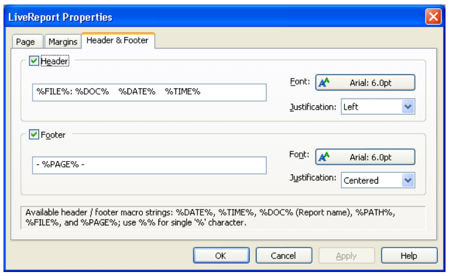
Use the LiveReport Margins Properties page to change the margins of a LiveReport.



1. **Top, Left, Right, Bottom** - The margin widths to use for the page. The margins are shown by a light-gray, non-printing box; the box can be hidden using the eye toolbar button menu.
2. **Units** - The units used by LiveReport for all of its settings can be set on the Page tab.

Header and Footer Properties

Use the LiveReport Header & Footer Properties page to change the margins of a LiveReport.



1. **Header** - When checked, the header is enabled. It will print at the top of the LiveReport page. The text is completely customizable; strings such as "Company Confidential" may be used. Also, macro strings like "%DATE%" and "%TIME%" will be converted into the actual date, time, filename, etc.
2. **Font** - Sets the header font.
3. **Justification** - Determines the header horizontal justification (left / centered / right).
4. **Footer** - The footer works just like the header, but is printed at the bottom of the page.

Supported LiveReport Object Types

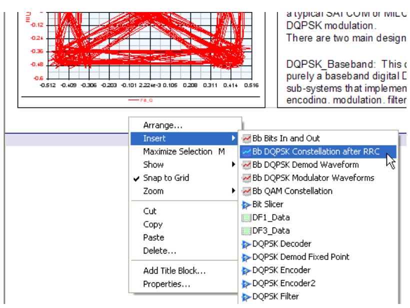
LiveReports supports most of the standard SystemVue object types.

Object Type	Supported?	Limitations
Graph	yes	A Graph has a single aspect ratio. If you have an open graph and a graph in report only the only the graph that is currently selected will own the aspect ratio.
Design	yes (partial)	Not supported - Parameters, PartList, and SubstrateSet
Notes	yes	
Datasets	yes	
Equations	yes	
Scripts	yes	
Tables	yes	
Analyses	no	Linear, Transient, ... have no view
Evaluations (sweeps)	no	Evaluations have no view
Syntheses	no	Syntheses have no view
Substrates	no	Substrates have no view

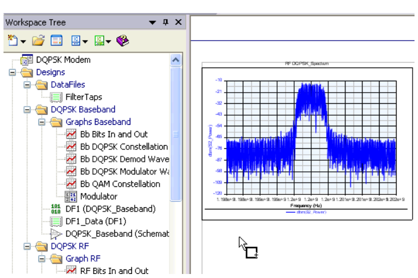
Adding a View Window to a LiveReport

There are two ways to put objects (windows) on a LiveReport.

- Right-click the page and select Insert then select one of the objects listed to insert a window with that object.

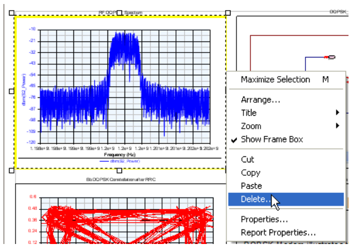


- Drag-drop an object from the workspace tree into the page using the mouse left button.



Removing a Window from a LiveReport

Select the surrounding rectangle (click it or select multiple with the select tool and draw a box) and then either click the Del key or right-click the mouse and select Delete... from the menu.

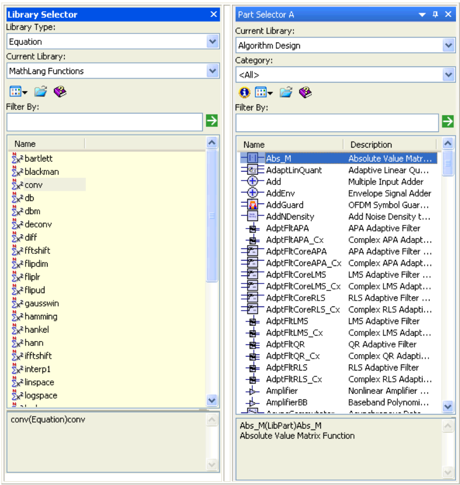


Managing Libraries

Libraries serve as a container for parts, designs, equations, and lots of other SystemVue objects. They let you keep all of your objects in one place, which makes it easier for you to organize the contents. SystemVue provides a number of libraries for your convenience and allows you to add custom libraries. Libraries that are added will be auto-loaded when SystemVue starts.

There are two dialog boxes in SystemVue that allow you to interact with Libraries: The Library Selector and the Part Selector. There is also a Library Manager dialog box that allows you to do things such as import and remove libraries.

The two dialog boxes that enable the interaction with objects are the Library Selector and the Part Selector.



The Library Selector allows interaction with all object types except parts, since the Part Selector is used to interact with Parts. You can think of the Part Selector as a specialized Library Selector where the Library object type is Parts. Both the Library Selector and Part Selector allow you to bring objects into your workspace or view objects that you have put into them from the workspace. A search text box is provided in both selectors to help you find objects in your libraries.

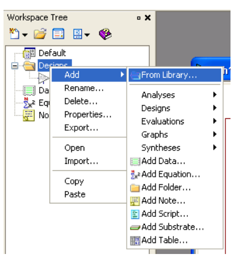
Contents

- *Using the Library Manager* (users)
- *Creating Custom Libraries* (users)
- *Adding Library Items to Your Workspace* (users)

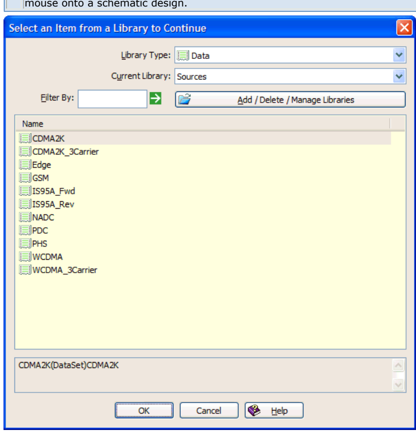
Adding Library Items to Your Workspace

Any object in a library can be added to your workspace. In the case of Symbols or Models you can just double-click (or edit) the object in the Library Selector. You can use the library selector to add any object type into your workspace.

If you don't want the docking library selector to be visible (taking up screen real estate) use the Add From Library Option as seen below.



Parts are special. They are not added (separately) to your workspace, but instead, are placed using a mouse onto a schematic design.



To Insert an Object from a Library into your workspace

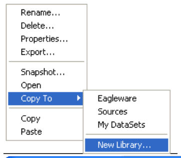
1. Select **From Library...**
2. Set the **Library Type** to the type of object you want to insert in your workspace
3. Set the **Current Library** to the Library you want to add from
4. Double-click the specific Object you want to add.

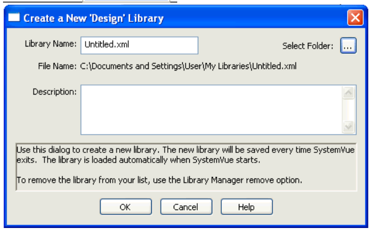
Creating Custom Libraries

A custom library can contain custom parts or designs (models and symbols and circuits), custom C++ data flow models, or anything else. Each Library Manager section can only hold custom libraries of its specific type (so you can not use a design library in the part selector, you can not use a Dataset library in the design selector, and so on). Custom C++ libraries must be created using the C++ model builder, detailed in the *Creating a Custom C++ Model Library* (users) documentation. To create libraries for other types, follow the procedure detailed below:

To create a custom library

1. Right click an object in the workspace tree or for a part right click a part in the Part Selector
2. Select New Library... from the Copy To menu



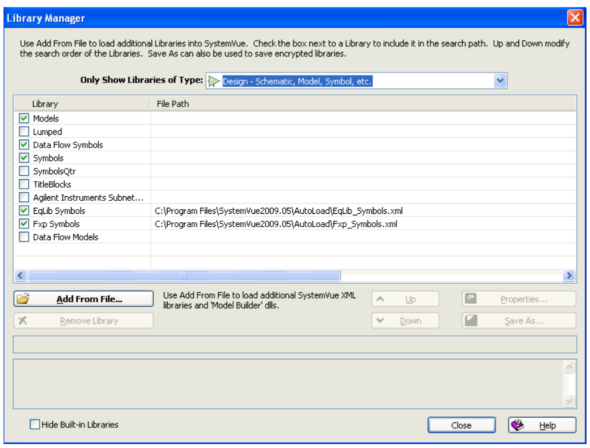


3. Set the library name.
4. If you want, browse for a different path for the new library. We recommend the My Workspaces folder for libraries, though.
5. Click **OK**.

Using the Library Manager

To open the Library Manager window:

- Click the Library Manager ( ) button on the Part Selector or Library Selector toolbar.
  or
- Select the **Tools** menu and **Library Manager**.



- **Only Show Libraries of Type** - Selects which type of libraries to view in the main window.
- **Add From File...** - Add a SystemVue XML library,
- **Remove Library** - Remove the selected library from the Library Manager.
- **Up** - Move the selected library up one position on the Library list.
- **Down** - Move the selected library down one position on the Library list.
- **Properties...** - View the properties of the selected library.
- **Save As...** - Save the selected library as some other name, or as a encrypted library.
- **Hide Built-in Libraries** - Hides all built in libraries from the library list. Only vendor and custom libraries are displayed.

You can use the Library Manager to:

- Add a Library from a File
- View Libraries of Different Types
- Add Libraries to the Search Path
- Remove a Library
- Edit the Properties of a Library
- Export an Encrypted Library

Add a Library from a File

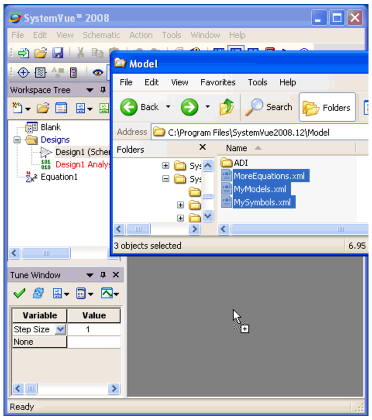
Select the **Add From File...** button to add a SystemVue XML library, a C++ custom library, or a wireless library to SystemVue. SystemVue ships wireless libraries in \AutoLoad folder and a large number of ADI models in Model\ADI folder of the SystemVue installation directory.

Adding XML Libraries

1. Click the **Add From File...** button.
2. Browse to the folder with the XML library you want to use.
3. Select one or more libraries (use Shift+Click, Ctrl+Click, or Ctrl+A to select more than one).
4. Click **Open** to add the library or Libraries to the available Libraries.

Adding XML Libraries via Drag-Drop

- Find the XML libraries you want to add using Windows Explorer. Select all of the libraries and drag then drop them into the SystemVue work area.



In the image above we add 3 new libraries (equation, model, and symbol library) to SystemVue.

Adding C++ Custom Libraries

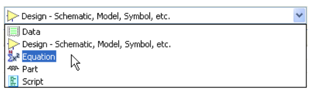
1. In the Library Manager, select the **Add From File...** button.
2. Set the Files of Type field to **SystemVue DLL Libraries**.
3. Browse to the folder with the DLL library you want to use.
4. Select one or more libraries
5. Click **Open**

Adding Optional Libraries

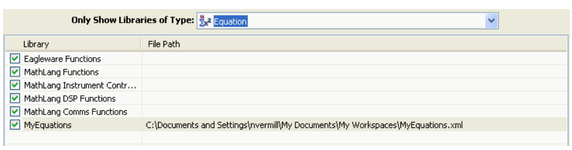
1. Use the above steps to add optional SystemVue libraries located in <SystemVue installation directory>\AutoLoad.

View Libraries of Different Types

The Library Manager lists all of the libraries that are currently loaded into SystemVue. Use the dropbox **Only Show Libraries of Type** to select which type of libraries to view in the Library Manager.



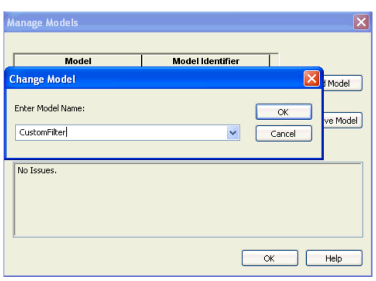
The selection Design - Schematic, Model, Symbol, etc. shows all libraries that contain schematics, models, or symbols. Notice that if you change the Type to **Equation** only libraries of equations are shown in the Library Manager.



Adding Libraries to the Search Path

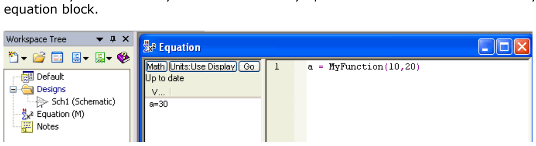
The checkbox next to each library determines whether or not the library is included in the search path. Once a library has been added to the search path, it will always be in the search path unless you manually remove it. Libraries at the top of the list have the highest priority in the search path. Use the **Up** and **Down** buttons to move libraries around in the list.

This feature is useful for libraries of custom models or symbols that you may have. Models and symbols from libraries that are included in the search path can be type in the Change Model or Change Symbol dialogs. For example, a library called MyModels that contains a model called CustomFilter has been added to the Library Manager and added to the search path. The model CustomFilter or any other model in MyModels can now be entered in the Change Model dialog directly.



Adding the MyModels library to the search path removes the need to type CustomFilter@MyModels or to add the model from the library to the workspace tree in order to use the model in a part.

Another helpful tip is to add custom Equation libraries to the search path. Functions in an Equation library that has been added to the search path can be called directly from any equation block. For example, a library called "MyEquations" that contains a function called MyFunction is added to the Library Manager and included into the search path. The function MyFunction or any other function in MyEquations can now be called from any equation block.



Functions in the MyEquations library do not need to be added to the workspace if MyEquations has been added to the search path.

Removing a Library

When you remove a library, you only remove it from the Library box in the Library Manager. The external library file is not deleted.

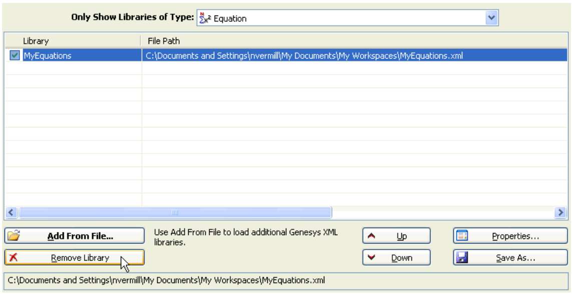
You cannot remove the Internal libraries. These libraries are read-only.

To remove a library from the Libraries list:

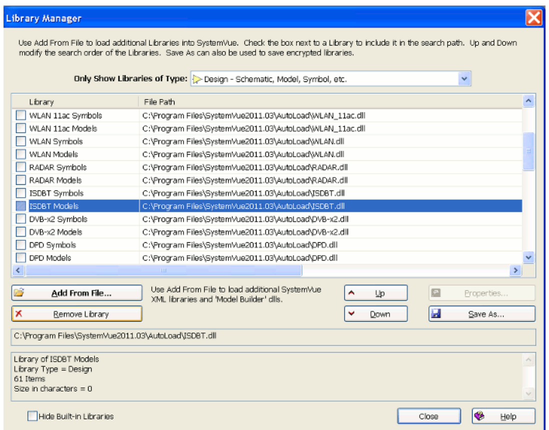
1. Click the name of the library in the Library Manager dialog.
2. Click **Remove Library**.

This does not delete the library file. It just assures that the library is not auto-loaded the next time you run SystemVue.

The following picture shows an example of removing a custom XML library.



The following picture shows an example of removing a wireless library.



Editing Library Properties

You can use the Library Manager to edit the properties of your libraries such as the name or description of the library.

To edit the properties of a library:

1. Click the name of a library in the Library list.
2. Click the **Properties...** button.
3. Make the changes you want, and then click **OK**.

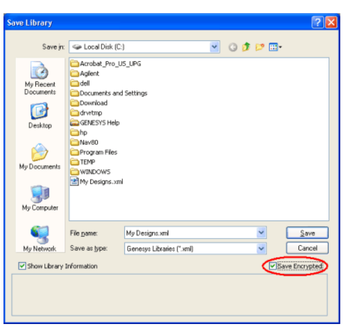
You cannot edit Internal libraries nor can you edit Encrypted Libraries. These libraries are read-only.

Export an Encrypted Library

SystemVue supports encrypting and using encrypted libraries. The option to encrypt a library is accessible through the Library Manager, which itself is accessible from the Library Selector (View/Docking Windows/Library Selector) or the the Tools Menu. Once you have created a custom library, you may save it as encrypted.

In the Library Manager, select your custom library and click the **Save As...** button on the right.

In the Save As dialog box, there is a checkbox labeled **Save Encrypted** on the lower right. Check this box as shown here:



The library will then be saved as encrypted. If the library you are encrypting is a Design Library, you will not be able to view the contents of the designs' part lists or equations.

Nets, Connection Lines and Buses

In a data flow schematic, data move directionally from input ports to output ports through parts and from output ports to input ports through connection lines. A connection line may be drawn by using the toolbar button to initiate connection line drawing mode or simply by hovering over a part's terminal until the cursor changes to connection-line mode, at which point you can click and drag to draw a connection line.

Contents

Connecting Parts in SystemVue

Connection lines are used to connect part terminals and other connection lines. If a part terminal or connection line end is unconnected, the arrow tail or tip is marked with a pink dot. The pink dot disappears after a connection is made.

To draw a connection line:

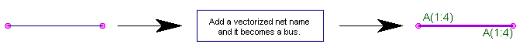
1. Click one of the two **Draw Connection** buttons from the main toolbar:
2. Click and hold the the start point on the schematic and drag the line to its end point on schematic.

OR

1. Hover over a part or connection line terminal and see the mouse cursor change into connection line mode. Click and drag the connection line.

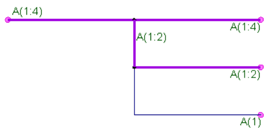
To create a bus:

1. Draw a connection line.
2. Double-click the connection line OR Right-click on the connection line and select **Net -> Edit Net Name...**
3. Enter a bus name to the new name text box. See *bus names*.
4. Click OK.
5. The bus name should label the connection line. The bus connection line changes to purple and is drawn as a thicker line.

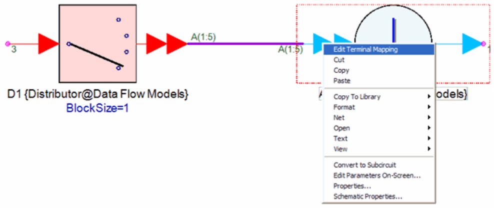


To tap connection line(s) off the bus:

1. Draw the bus tap connection line.
2. Double-click the connection line OR Right-click on the connection line and select **Net -> Edit Net Name...**
3. Enter the bus base name and the indices for the bus connection lines you want to tap. See *bus names*.
4. Click OK.

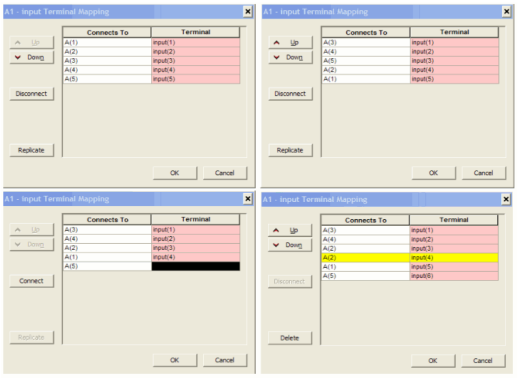


To edit terminal mapping: The precise mapping of individual line(s) of a bus to a multi-input port component can be achieved using the **input Terminal Mapping** or **output Terminal Mapping** dialogs which is invoked by right clicking on the multi-input or output pin and selecting **Edit Terminal Mapping** as shown.

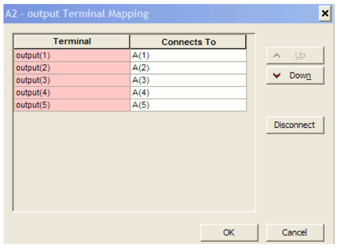


1. Use the **Up** and **Down** buttons to rearrange the incoming bus line(s) with respect to the input sequence. Note that only incoming lines may be moved with respect to the input ports.
2. Use the context sensitive **Disconnect** and **Connect** toggle button to delete and establish connections. Once a line is disconnected, it is automatically moved to the bottom of the queue. Connecting it now establishes a link between the last input part.
3. Use the context sensitive **Replicate** and **Delete** toggle button to replicate an incoming line to drive an additional input to the component. By default, this additional input is inserted immediately below the line entry that was replicated, resulting in a duplication of the **Connect To** entry and an occupation of the next port index on the component side, resulting in all successive inputs being assigned higher indices of input port number than before.

The following four figures show the default view of the input terminal mapping dialog box, the change of listing because of movements up and down the sequence, followed by the disconnection of A(5) and reconnection followed by insertion of a duplicate of A(2). Note how the port side now has input indices ranging from 1 through 6, whereas the connection side has a duplicate of the second incoming line. Note also how the previous association of A(1) and input(4) has now been replaced by one between A(1) and input(5) and so on.



The output terminal mapping dialog is a vertical mirror of the input terminal mapping dialog except that it does not have the ability for replication and deletion of duplicate entries. Unilateral input-side replication of connections provides a barrier against proliferation of bus line(s) at the output of the transmitting component and transfers the responsibility of duplication to the input of the receiving component(s).



Connection Line Net Labels

Connection lines by default have no net label, so they inherit the automatically assigned net of part terminals or other connection lines that they are connected to. If a connection line is given a Net Label, then that label becomes the net that the connection line resides on.

A net label could be a simple name or number which represents a single net, or it could be a Bus label, in which case the connection line represents several nets. Bus labels are simple names or numbers followed by indices specified in parentheses. The syntax is as follows:

BaseName(Start:Stop:Step)

where everything except the BaseName is optional.

Note that the Start:Stop:Step ordering for Bus labels is different than the Math Language range vector ordering of Start:Step:Stop. This was done in order to conform to the industry standard bus notation ordering.

Here are some examples:

Net Label	Nets, in order
MyNet	MyNet
MyNet(3)	MyNet(3)
MyNet(1:3)	MyNet(1), MyNet(2), MyNet(3)
MyNet(2:1)	MyNet(2), MyNet(1)
MyNet(0:4:2)	MyNet(0), MyNet(2), MyNet(4)

You may use variables or expressions for each of Start, Stop, and Step in the bus indices. This results in a dynamic bus width: When the variable(s) you use change, so do the widths of buses that use them.

To assign a net name:

1. Double-click on the connection line OR Right-click on the connection line and select **Net -> Edit Net Name...**
2. Enter a net name and click OK.

To remove a net name:

1. Double-click on the connection line OR Right-click on the connection line and select **Net -> Edit Net Name...**
2. Delete the net name from the field leaving the field blank and click OK.

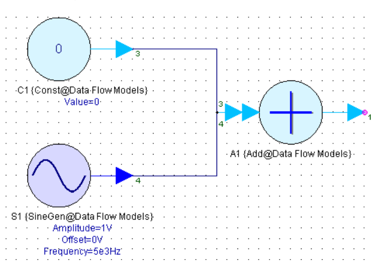
If two connection lines share a common net from their Net Label, but they do not look visually connected, they are still connected for simulation purposes.

Renaming a connection line can cause redefinition to a bus or to a simple connection line, e.g. A to A(1:4) or A(1:4) to A.

When an unconnected connection line is created, it does not have a net name. When the connection line is connected to net, it may gain a net name from the net. If the net has context and the net name is not set, an implicit net name is generated which may change with the schematic. This net name is an integer that is shown at the ends of the net. When a net name is explicitly assigned, the net name becomes persistent. While a persistent net name can be an integer, begin the net name with an alphabetic character.

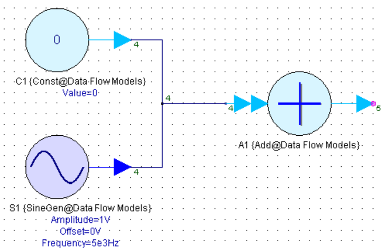
Connection Lines and Ports

If no Net Labels are given to connection lines, they are automatically assigned nets in an intelligent manner, taking into account directionality and type (standard or bus) of ports they are connected to. Connection lines that end at a Bus port will produce separate nets at that bus port, as shown here:



Notice in the above schematic that the output from the Constant source is on net 3, while the output from the Sinusoidal source is on net 4. This is the preferred way to connect multiple things to a Bus port since it produces separate nets for each terminal connected to it. This allows an ordering to be defined at the Bus port via the Terminal Mapping dialog box, which will be discussed shortly.

Since the Addition operation performed by the adder part is commutative, ordering of the inputs is irrelevant, so the following schematic would yield the same results:

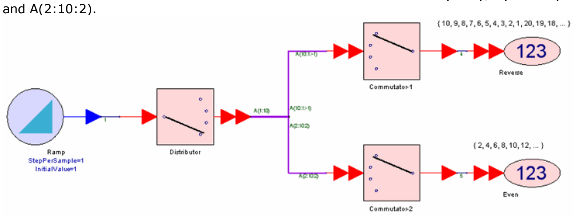


In the above schematic, both the Constant source and the Sinusoidal source are on the same net. The simulator is intelligent enough to expand the net into 2, since 2 outputs are feeding an input, however the ordering is undefined.

Connection Terminology

A **connection line** is a drawn line on the schematic that can be used to connect an output port with an input port. A **bus** is a connection line that is a collection of two or more connection lines. A **net** is a group of simple connection lines that share a unique **net name** and a common value at any instant.

The following schematic has 13 nets with net names: 1, A(1) ... A(10), 4 and 5. Net 1 is a simple connection line between the Ramp and the Distributor part. Similarly, Net 4 and 5 are simple connection lines. Net A(1) connects an output of the Distributor to one input of Commutator-1. Net A(10) connects an output of the Distributor to input ports of both Commutators. All nets named A are contained in the 3 buses labeled A(1:10), A(10:1:-1) and A(2:10:2).



Mapping Nets to Ports

The mapping from connection line nets to a particular part port (terminal) can be seen and modified by looking at the Terminal Mapping dialog, accessible by right-clicking on the terminal and selecting the *Edit Terminal Mapping* menu entry, if it exists. The menu entry will not exist if there is no ambiguity in the ordering of the net to part terminal mapping, as is the case when there is a single net connected to the part terminal.

The Netlist for a part (all terminals) can be viewed in the Netlist tab of the Advanced properties of a part. See *Part Properties* (users) for details.

There is no net name or ordering ambiguity in a connection between a standard port and a simple connection line.

However, this is not the case for a bus entering or leaving a bus port. The default assignment for the bus port matches the order of ports with the order of the net names. For example in the schematic shown under the *Connection Terminology* heading, the bus port of the Distributor has ports named output(1:10), i.e. output(1) through output(10), which is mapped to the bus connection line nets named A(1:10), i.e. A(1) through A(10).

The default mapping may not be what is intended, so it is possible to specify an arbitrary ordering, replicate, and disconnect the sub-nets. These actions can be performed by means of the Terminal Mapping dialog. To access it, right-click the part terminal you want to define the terminal mapping for and select "Edit Terminal Mapping" if that menu item exists. If it does not exist, there is no ambiguity in ordering of the nets presented to the terminal.

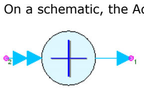
The Terminal Mapping dialog displays differently depending on whether it is for an input port or an output port. For the input Terminal Mapping dialog, nets are in the left column (Connect To) and ports are in the right column (Terminal). For the output Terminal Mapping dialog, nets are in the right column (Connect To) and ports are in the left column (Terminal). When a row is clicked, the net for that row is selected for operation by the Up, Down, Connect/Disconnect or Replicate buttons. Click the desired button for the following results:

1. The Up button will swap the selected net with the one above it.
2. The Down button will swap the selected net with the one below it.
3. The Disconnect button will disconnect the select net and decrement the number of sub-nets.
4. The Connect button will reconnect the selected (disconnected) net with an appended sub-net.
5. The Replicate button will present a duplicate sub-net to the port which is appended to the list.

To exit the Terminal mapping dialog, click the OK button to save the changes or the Cancel button to discard the changes.

Part Ports (Terminals)

On a schematic, the Adder part is depicted as follows.



This part has one input port with net name 2 and one output port with net name 1. The input port is a bus port (indicated by the two arrows), while the out port is a standard port (indicated by a single arrow). A bus port is an ordered set of ports that can be expanded dynamically. In other words, while a single port resides on a single net, a bus port can reside on multiple nets. Bus ports are described in more detail below.

Every part port is assigned a unique net name when placed unconnected on the schematic. Inputs are distinguished from outputs, because input port arrows point into the part symbol. For additional visual cues, see *port data type* (sim).

Parts, Models and Symbols

Contents

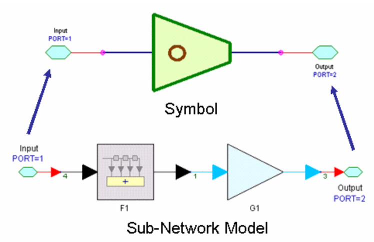
Finding Symbols and Models during Simulation

During an analysis or code generation , the models are instantiated for each part. The model instantiated is determined in the following order:

- If a *design configuration* (users) is used, all parts are switched to the model specified.
- Then if the model was not specified in a design configuration, the model will be instantiated based on what is selected in the part's *manage models list* (users).
- If the model has it's *path specified* (users) (e.g. Model1@Library1), then it is instantiated from the library specified.
- If the model does not have a path specified, the *library search path* (users) will be used to find the model.

Mapping Symbols to Models in Parts

Port names and numbers used on **symbols** map the symbol terminals to the sub-ciruit or model nets. Both the symbol and sub-circuit or model contains ports. The mapping **precedence** if first by **port name** and then **port number**. If the **port names** match between the symbol and the sub-circuit or model then these **port names** will be used and **port numbers** will be ignored.



Note
It is important to recognize that the mapping is taking place between the **ports names and numbers** used on the **symbol** and **port names and numbers** used to define the **sub-circuit or model**.

Caution
Net names on a part placed in a schematic provide no useful information as to what the **symbol** port names or numbers are. The user must open up the symbol which represents the sub-circuit or part to determine what the port names and numbers are actually mapped to.

If the symbol port naming or numbering is wrong, the model will be incorrectly connected in the simulation.

Models

A single part can support multiple models. Models can even be different types. Supported models types are:

- ***Math* (users)**
- ***Code* (users)**
- ***Sub-Network Models* (users)**

When a model is changed any common properties from one model to the next are copied over to the new model. Furthermore, the old model is cached so if the user decides to return to the old model they won't need to re-enter the parameters.

Models can be saved in libraries or in the workspace tree. The model naming convention is: **ModelName@LibraryName**. Local model versions can be copied from an original model in a library. By default these local model copies have the same model name as the parent model in the library.

For more information on models see **User Defined Models** under the **Using SystemVue** section in the users guide.

Tip: Use the toolbar Show/Hide (eyeball) button to show or hide all the model names on a schematic.

Hint: During a simulation a model appearing in the workspace tree will always be used before a model in a library even if the library name has been specified for the model.

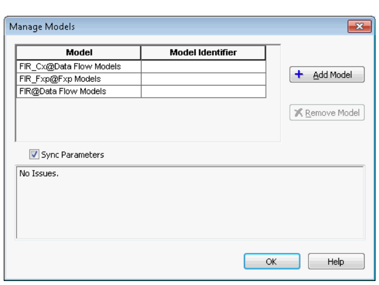
Note: A part can have several models and each model can have its own set of parameters. Those parameters with the same name and type can be synced, i.e. a change in value for a synced parameter is a change for all models that have this parameter.

Changing a Model

Each part can contain several models. Pick one using the Model combobox in Part Properties.

Note: Certain specialized symbols, like those using %MACROS%, are designed for use with certain specific models. When you change a model, you may occasionally also need change the part's symbol, since the symbol might no longer match.

The list of available models can be changed using the model manager. Click on the Manage Models button (Manage Models...) to bring up the model manager dialog box.



Model Identifier - This parameter is used to distinguish between models of the same name.

Sync Parameters - This check box declares if the common parameters will be synced between the models.

Status Pane - This pane will alert users to potential errors or warnings.

To add a model:

1. Click on the Add Model button (Add Model)
2. Select the desired option:
   1. (A **list of models** in the workspace are listed)
   2. **From Library** (load a model from a library)
   3. **Enter Model Name** (select a model name)

To remove a model:

1. Click on the model to be removed
2. Click on the Remove Model button (Remove Model)

Defining Configurations for a Design

A design configuration tells an *Analysis* (sim) or a Code Generator to use specific model for a part which is already included in the manage model list of that part. For more information read *Configurations* (users) in *Modifying a Design* (users) documentation.

Creating a Model

To create a model select the type of model to be created. Follow those instructions:

- ***Math* (users)**
- ***Code* (users)**
- ***Sub-Network Models* (users)**

Parts

A **part** is a the fundamental building block in any schematic. Each part contains both a **Model** and a **Symbol**, which, for maximum flexibility, may be changed independently.

Only **parts** can be placed on schematics. The part's **symbol** is the *image* on a schematic and the part **model** is *what is being simulated*. Users connect parts together on a schematic by placing wires between the part's symbol terminals. These *connection points* are called **nets**. The part itself maps symbol terminal pins to the model nets which are what actually gets simulated.

Double-click a part to access **Part Properties**, which provides a quick way to identify or change:

- The visible symbol - via the Advanced Settings button
- The model being simulated
- The model's parameter settings and
- Other part characteristics
- Furthermore, every part has the ability to ignore the parent model – which can **short** all simulation nets together or make all part nets have an **open** connection.

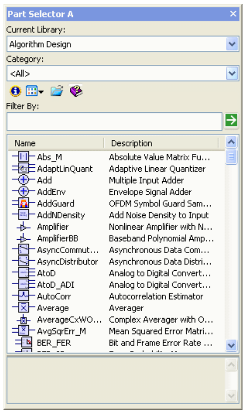
Each part supports a list of multiple models, with a means to manage these models in Part Properties.
Parts, their models, and symbols can be saved in libraries for reuse.

Placing Parts on a Schematic

From the Part Selector

To place a part from the part selector:

- **Click** on the part in the part selector.
- **Move** the mouse over the schematic. The mouse cursor will change to a plus sign when placed over the schematic.
- **Click** the schematic where the part is to be placed.



From the Keyboard

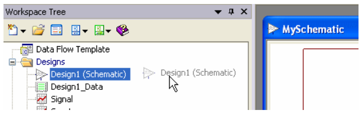
Certain frequently-used parts can be placed via the keyboard. Inside SystemVue, use the Help / Keystroke Commands menu to display Appendix A, which lists the available parts.

From the Workspace Tree

Schematics can be dropped into other schematics to create a sub-network model. Models, and S-Parameter files can be dragged and dropped onto the schematic. When a schematic or model is dragged and dropped on a schematic a sub-network model is created along with a generic symbol. When an S-Parameter file is dragged and dropped onto a schematic the dataset part will placed on the schematic.

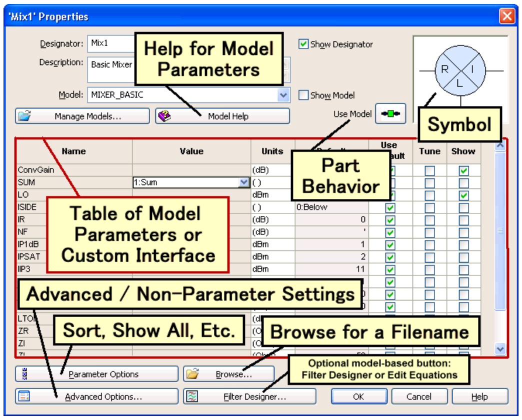
To place a part from the workspace tree:

- **Click** on the schematic, model, or S-parameter dataset.
- **Move** the mouse over the schematic. The mouse cursor will change to a plus sign when placed over the schematic.
- **Click** the schematic where the part is to be placed.



Part Properties

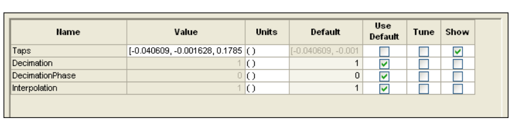
Each part has the following characteristics:



- **Designator** - Descriptive text that appears on the schematic that references the part.
- **Show Designator** - When checked the designator will appear on the schematic.
- **Description** - Documentation info for the part. This info can be displayed in the part selector.
- **Model** - Name of the model to be simulated. The format is **ModelName@ LibraryName**. From this combo box the user can select the **active model**. The **model parameters table** will automatically be updated with this selection.
- **Show Model** - When checked the model name will appear alongside the designator on the schematic.
- **Manage Models** - When clicked will open a dialog box giving the user the ability to manage the models that are available for selection.
- **Model Help** - When clicked will open the help page providing descriptive information for all parameters in the model parameter table.
- **Part Behavior** - This button controls the behavior of the part. The four options are:
  - **Use Model** - Use the currently specified model ( ). This is the default state.
  - **Disable, Open** - All model net connections are opened ( ). No data will flow through this model.
  - **Disable, Short** - All model net connections are shorted ( ). The model is bypassed.
  - **Control by Equation...** - Use an equation expression to control the Part Behavior. The expression must evaluate to 0 = Use Model, 1 = Disable to Open, 2 = Disable to Short. For example: Enter UseMyModel, click OK, and add this equation (either to the design containing the part or as a global equation) **UseMyModel=?1**, then click the Go button at the top of the equations window. Now you can Tune UseMyModel to use/open/short the part. You can enter ANY equation in the Control by Equation window and as long as it evaluates to 0, 1, or 2, the part will use the appropriate setting. To later modify the equation, simply click the Part Behavior button again and select Control by Equation.
- **Symbol** - Shows a picture of the symbol associated with the part. This symbol can be changed on the **Advanced Options** dialog box.
- **Models Parameters Table** - This table contains the list of parameters specified by the model. In some cases there are models that have a custom interface that appears in the same area of the dialog box. See the model help for specifics on these models.
- **Parameter Options** - Click this button for options like sorting all model parameters in the table alphabetically, checking all the Show checkboxes, etc.
- **Browse** - This button will be enabled when the user clicks on a model parameter that needs a filename. (If the model has no filename parameters, the button will be hidden.) The user can then browse to the desired file.
- **Advanced Options** - Gives the user the ability to change / create a symbol. Change its positioning and manage the mapping of the **symbol terminals** to the **model net**. (See details on each Advanced Options tab page below.)
- **Filter Designer** - This button is only available for parts whose model is a filter; click to close this dialog box (with an OK) and bring up the Filter Designer GUI
- **Edit Equations** - This button is only available for parts with equations (like MathLang and Sink); click to close this dialog box (with an OK) and bring up the equations editor window.

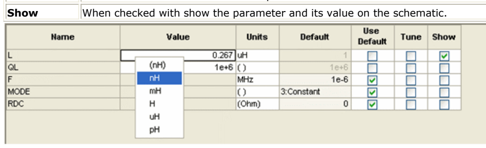
Model Parameters Table

This table lists all model parameters, their values, units, and characteristics.



Name	Value	Units	Default	Use Default	Tune	Show
Taps	[-0.040609, -0.001628, 0.1785	( )	[-0.040609, -0.001	☐	☐	☑
Decimation	1	( )	1	☑	☐	☐
DecimationPhase	0	( )	0	☑	☐	☐
Interpolation	1	( )	1	☑	☐	☐

Column Headings	Description
Name	Parameter name specified in the model. (Read-only)
Value	Parameter value. The values can be in following forms: numeric, enumeration, variable, or formula. Allowed enumeration values are specified by the model. A formula or an equation can be used in an enumeration field.
Units	Determines the units the parameter value is interpreted in.
Default	This is the default parameter value specified in the model. (Read-only)
Use Default	When checked the default model parameter value will be used.
Tune	When checked will make this model parameter value tunable.
Show	When checked with show the parameter and its value on the schematic.



Changing the units in the units drop down will NOT do unit conversion. If you want the value converted to a new unit you need to right click on the parameter's Value or Units field and select the new unit. An example right click menu is shown above.

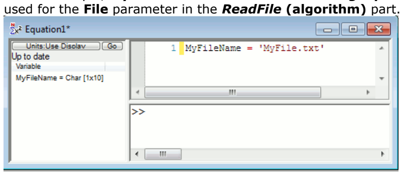
For example, in the picture above when changing the unit for L from uH to nH:

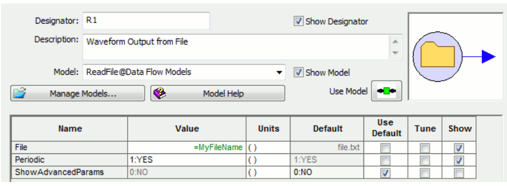
- if you use the right click menu the value will change from 0.267 to 267.
- if you use the normal drop down unit menu the value will remain 0.267.

Use MathLang Variables for Parameters

MathLang variables can be used to pass values to parameters. If the parameter value is not text, simply type the MathLang variable name into the parameter's value field and the variable's value will be passed on to the parameter.

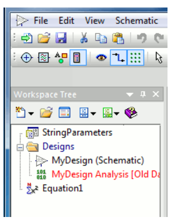
However, if the parameter value is text (e.g.file name, or a text string, etc.), use the syntax **=MathLangVariableName** to fill the parameter's value field as shown in the following example.
In this example, **MyFileName** is defined in a ***MathLang Equation* (users)** page, and is used for the **File** parameter in the ***ReadFile* (algorithm)** part.





⚠ When using MathLang variable to pass parameter values, make sure the MathLang variable is defined in a **scope** that is accessible by the schematic. If the MathLang variable is not accessible by the schematic, it will be treated as undefined and will result in errors.

The easiest way to **understand scope** is to look at the relative position of the ***MathLang Equation* (users)** (that defines the variables) and the schematic (that uses the defined variables) on the workspace tree. The ***MathLang Equation* (users)** can **not** be farther from the root of the tree than the schematic. In the example below, notice that **Equation1** where the MathLang variables are defined is at the same level from the root **StringParameters** as the **Designs** folder, where the **MyDesign** schematic (in which the MathLang variables will be used) is hosted. This makes **MyDesign** one level farther away from the root **StringParameters**. It is fine to move **Equation1** into the **Designs** folder on the workspace tree, which makes **Equation1** and **MyDesign** on the same scope level.

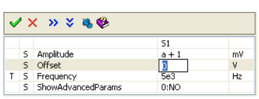


Editing Part Parameters On a Schematic

Part parameters that appear on the schematic can be directly edited without opening up the part properties dialog box.
To edit the part parameters:

- Move the mouse pointer over the part text on the schematic. Note that the mouse pointer changes to resemble an I-beam (the text edit cursor). Click in the text.

The following editor will appear:



Things you can do when editing a single part:

- Type a new value
  - Click outside the box or click Accept to close/accept the changes
  - Click a different part to Accept and switch parts (if pinned)
- Click in the S column to set a parameter show/hide
- Click in the T column to set a parameter tunable/fixed
- Click up/down arrows to edit other parameters
- Use a button to do more

The buttons on top:

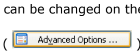
Accept	Do an OK
Cancel	Cancel all changes
<< and >>	Expand and contract the box to show/hide the Tune and Show columns.
Up and Down	Expand and contract the box to show/hide non-shown parameters.
Pin / Unpin	When pinned, clicking another part will move the box. When unpinned, the box just closes (Accept).
Help	Brings up this help

Keys Supported:

Up Cursor	Move to prior value
Down Cursor	Move to next value
Tab	same as Down Cursor
Shift+Tab	same as Up Cursor
Enter	Accept
Esc	Cancel

Advanced Options

Part symbols and part connectivity can be changed on the advanced options dialog box.

Click the Advanced Options button () to bring up the Advanced Options dialog box.

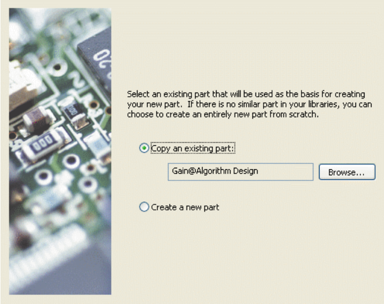
See individual tab page topics below for additional information.

Creating a Part

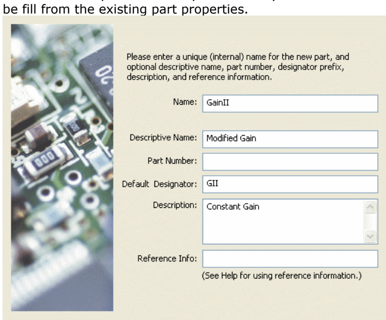
When the user has a **model** and **schematic symbol** they want **combined into a part** they can use the Create Part Wizard to automate this process. The finished part must be placed in a library for future reuse.
To create a part using the Create Part Wizard:

1. Click **Action** on the menu and select **Create Part Wizard**.

2. Browse for an existing part to use as a starting point or begin with a blank part.

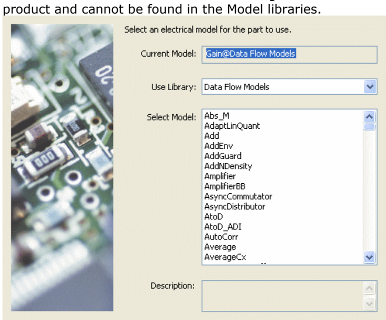


3. Click **Next**.
4. Fill in the descriptive fields as you want. If you re-used an existing part, the fields will be fill from the existing part properties.

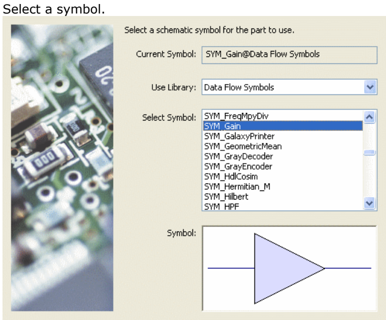


Note: The RefInfo string is composed of |-delimited "name|reference-link" pairs of substrings. Each pair consists of a menu item and a command, usually a URL, directory path, or file (.doc, .txt, .htm, etc.)

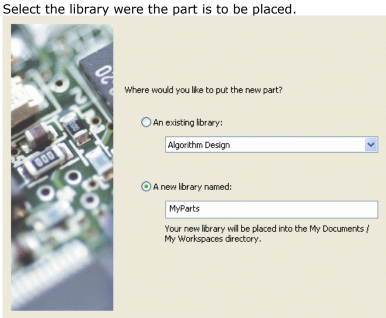
5. Click **Next**.
6. Select the model to use. Note that the <registered> models are internal to the product and cannot be found in the Model libraries.



7. Click **Next**.
8. Select a symbol.



9. Click **Next**.
10. Select the library were the part is to be placed.



11. Click **Finish**.
12. Follow dialog prompts to add the new part to a library.

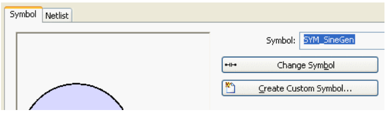
Symbols

The part **symbol** is the graphical picture the user sees on the schematic that represents the part. Symbols can easily be created, modified, or changed for a given part.

Changing a Symbol

To change a symbol:

1. Click the Advanced Options button (Advanced Options ...) on the part properties dialog box.
2. Click the Change Symbol button (Change Symbol)
3. Select a new symbol or option
   1. (A **list of symbol names** in the workspace are listed)
   2. **From Library** (load a symbol from the library)
   3. **Edit Symbol Name** (change the name of a symbol)



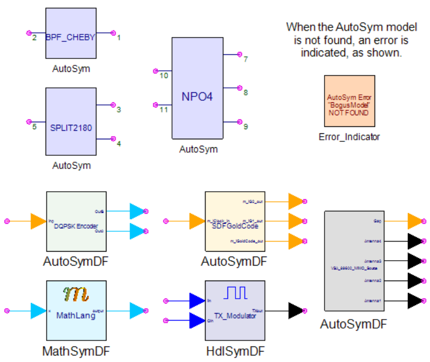
Algorithmic and Automatic (Dynamic) symbols

"Algorithmic" symbols are those that are created automatically by SystemVue, as opposed to being hand-drawn and stored in an XML Symbol Library. These symbols are defined using a root name / modifier format. Usually the modifier is a simple number 'N' representing the number of ports, switch-throws, etc. Here's a list of commonly-used algorithmic symbols:

- **SwitchN** – switch with 'N' throws
- **SplitN** – an N-way splitter
- **Box-M-N** – a filled rectangle w/ M pins on left and N pins on right
- **N-XFile** – X-Parameter File
- **N-XData** – X-Parameter Dataset
- **N-XFile-Gnd** – X-Parameter File with ground
- **N-XData-Gnd** – X-Parameter Dataset with ground

Note: These parts are normally built for the Genesys-standard schematic grid spacing of 1/6th inches. To generate symbols for an ADS-standard 1/8th grid, append the @SymbolsQtr suffix to the symbol name.

"Automatic" symbols are a subset of algorithmic symbols, which are based on a **model**. The symbol is a filled box with terminal pins on the left (input) and right (output); if in/out is not specified, the pins will be split evenly between the 2 sides. In addition, model port info is used to label the symbol pins.



Automatic Symbols are specified as follows:

1. **AutoSym** – A Genesys symbol (RF part); used for Spectrasys schematic symbols.
2. **AutoSymDF** – A SystemVue "Data Flow" symbol, with arrowheads on the I/O pins.
3. **MathSymDF** – Just like AutoSymDF, but with the MathLang gradient 'M' icon at the top.
4. **HdlSymDF** – Just like AutoSymDF, but with a blue square wave at the top to indicate an HDL part.

- Note that if the symbol cannot find the specified model, an error is shown as indicated.
- Data flow pin colors are based on the model port info.
- Parts with only 1 input or 1 output will be drawn as circles or ellipses, depending on the situation.
- Unambiguous terminal pin text will be omitted.
- The box fill color is based on the average of all the pin colors. In Spectrasys, since the pins are always dark blue, the fill color for AutoSym will always be light blue.
- An *optional* model suffix may be specified; for example, use **AutoSym-MyModel@MyModelLibrary** to fully specify the model. The @Lib is optional, but can only be omitted if the model can be found without it.
- An *optional* icon can be placed on the symbol by appending an icon (subsymbol) name to AutoSym or AutoSymDF. The icon must be in a loaded symbol library and the icon name must be inclosed within curly-braces { and }. Valid icon names include {MathLangM}, {VSA Icon}, and {RfLink Icon}. The model suffix (if any) must come AFTER the icon suffix.

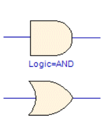
Dynamic Symbol Switching

This is an advanced feature, for use by expert users.

The symbol associated with a part can be changed *dynamically*, based on a part parameter. This is done via string substitution (macros); the macro substitution character is '+'.

For example, suppose there is a part called "RotarySwitch", which has a parameter "Throw". If the part's symbol is Switch+Throw+, when Throw=0, the symbol used by the part will be Switch0; likewise, if Throw=3, the symbol will be Switch3. If there is no symbol which matches the computed name, a placeholder "Not Found" symbol will be displayed.

In addition, the *names* of enumerated parameters can be used in the computed symbol names: Suppose there is a MathLogic part with a Logic paramerer, which is an enumeration (And = 1 and Or = 0). Like the previous example, if the part's symbol is Sym_+Logic+, when Logic=1, the symbol used by the part will be Sym_1. However, if the enumeration prefix character '#' is used, the NAME of the enumeration value will be used to build the symbol name: Sym_+#Logic+ will become Sym_And, which is a lot more clear when these symbols are used in a library.



Create a Symbol

To create a symbol based on an existing part:

1. Right click the part and select Open / Symbol.

2. Modify the new symbol
3. Optionally, double-click the original part and change the symbol to the new custom symbol.

To create a new symbol "from scratch":

1. Click the New Item button ( ) on the Workspace Tree toolbar, then click "Designs", then select "Add Schematic Symbol"
2. Enter the symbol's name
3. Draw the symbol in the schematic area. Use the Annotation toolbar to place text, lines, arcs, and other drawing objects.
4. Place input (**i** key) and output (**o** key) ports where symbol terminals are to be located.

   Note: Ports do not appear on the schematic when the symbol is used in a part.

5. Connect the symbol to the ports. These connection points are the connection points seen on a schematic when a part is placed.
6. Change the **port designator** to give the **symbol terminals** a name. This name is used to map symbol terminals to model nets.

Displaying Parameter Values on a Symbol:

When any (not just an algorithic) symbol is drawn on a schematic, symbol text is processed prior to display, using a technique called "Macro Substitution". The text within the '%' characters will be replaced with the appropriate value. For example, Name=%Model% would be displayed as "Name=Resistor" on a symbol using a resistor model.

For example, when "Impedance = %L%" is drawn on a schematic, the value of parameter 'L' is retrieved from its model and the result is "Impedance = 1.5". Another common use is to place the model name on the symbol.

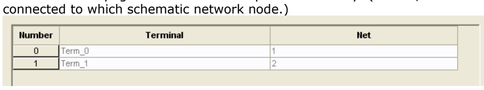
To use this advanced feature, place special "macro" strings in any symbol text:

1. Place a text annotation anywhere on the schematic symbol
2. Double-click it and change the text, so that it includes one or more macros from the table below.
3. Click OK

Macro	Result
%Model%	Name of the model attached to the schematic part
%MODEL%	Name of the model in UPPERCASE
%Des%	The part designator: R1, L3, etc.
%ParameterName%, where the name is any model parameter name, such as R, C, L, etc.	The actual value of the parameter.
%%	Displays a single % character.

Netlist Options

The **Netlist** tab page shows the current part connectivity. (That is, which terminal is connected to which schematic network node.)



Terminal - Names of the symbol terminals.
Net - Name of the schematic net the symbol is connected to.

Note: These fields are read-only unless there is no schematic. Connectivity is then determined by the names in the Net field.

Overview

A Data Flow simulation is used to understand a communication system at the algorithmic level using time domain analysis for baseband and RF signals. An RF analysis in Data Flow consists of the time domain analysis of the modulation information centered at the RF carrier frequency (commonly called the RF characterization frequency). The information or modulation bandwidth is based on the sampling frequency of the Data Flow analysis.

The RF analysis in SystemVue can be done using:

- RF Data Flow models OR
- Co-simulation with an RF architecture simulator using the **RF Design Kit**

The connection between Data Flow and RF simulator called Spectrasys is through a part called the *RF Link* (algorithm). This RF Link part is placed in a Data Flow schematic. When the Data Flow engine executes the RF design will be characterized at the carrier frequency with respect to frequency and power. This characterization information will be used by Data Flow to determine the output response.

See *Theory of Operation - RF* (users) for additional information.

RF Link Limitations

Use of RF (Spectrasys) designs in Data Flow schematics has certain limitations. These limitations include:

1. The paths through the RF design from any connected input to any connected output must **NOT** include any of the following models (the models below **may be used in LO paths**):
   - models that provide **frequency multiplication, frequency division, or analog to digital conversion**. These models include *FREQ_MULT* (rfdesign), *FREQ_DIV* (rfdesign), *DIG_DIV* (rfdesign) and *ADC_BASIC* (rfdesign).
   - variable gain amplifiers/attenuators. These models include *VarAmp* (rfdesign), *VarAmp1V* (rfdesign), *ATTN_Ctrl* (rfdesign).
2. The paths through the RF design from any connected input to any connected output must **NOT** go through any mixer LO port, that is, the LO signals for all mixers in the RF design must be provided in the RF design itself (they cannot be provided from the top level Data Flow design).
3. The *IR* (Image Rejection) parameter for mixers in the RF design is ignored.
4. **Interfering signals** created internal to the RF design are ignored. For example, in a poorly designed receiver combinations of the RF input signal mixed with a mixer LO may create intermods that fall in the desired channel bandwidth. This type of interference is being ignored.
5. Carrier noise includes both **amplitude** and **phase** noise. Only **phase noise** is modeled in the RF Link. The manifestation of amplitude noise on a carrier is seen as asymmetric noise centered around the carrier.
6. **AM to PM distortion is not currently supported** in Spectrasys models so these distortion effects will also be ignored in the RF Link.
7. All **noise** simulated in the RF Link is done at a **single temperature**.
8. **X-Parameter** models that translate frequency are not supported.

Simulation

Here are some things to consider when setting up an RF / Data Flow co-simulation.

Data Flow Specific

- The Data Flow input to RF_Link must be a *complex envelope signal* (sim) with a non-zero characterization frequency. This signal is typically defined by use of an *Oscillator* (algorithm), *Modulator* (algorithm) or *CxToEnv* (algorithm) model.

> **Caution**
> The speed of phase noise simulation in Data Flow is dependent on the offset frequencies and range of offsets being simulated. For example, a phase noise simulation that covers the offset frequencies from 100 Hz to 1 MHz will be much slower than an offset range of 1 kHz to 1 MHz.

RF Link Specific

- To enable thermal noise analysis in the **RF Link**, check the **Enable Thermal Noise** checkbox. When this checkbox is checked, the Spectrasys design characterization will include thermal noise from all parts that generate thermal noise. This includes thermal noise from passive parts and noise due to noise figure from active parts. However, thermal noise from the source model associated with the **Input Part** is not included. The RF_Link input noise is presumed to be already included in the input data flow signal. The noise analysis is performed over the RF Link frequency range.

> **Caution**
> In RF designs normally all RF ports generate thermal noise. However, in Data Flow simulations thermal noise is not modeled unless the user specifically adds appropriate Data Flow noise models. The **'Add source thermal noise to input'** option, when checked, will automatically add thermal noise to the Data Flow input signal driving the RF Link that represents the RF input port noise commonly modeled in an RF design. During an RF Link simulation the RF input port of the RF Link design does **NOT** generate thermal noise. Is is assumed that RF Link input signal contains the correct input thermal noise when the **Enable Thermal Noise** option has been checked. If the RF Link data flow input signal contains noise AND the 'Add source thermal noise to input' is also checked the total noise may be double counted.

- The input frequency characterization range of the RF Link is nominally divided into 101 equi-distant frequency points. This input frequency range is automatically converted within the RF design to account for frequency translation caused by mixers. Alternatively, the user can specify their own characterization frequency range.

- The RF Link supports mixer conversions to DC (0 Hz) and output baseband signals. Thus, ZIF (zero-IF) downconverter applications are supported. This includes any non-ideal isolation from LO to mixer input and mixer input to LO, which results in downconverter spectral products at 0 Hz.

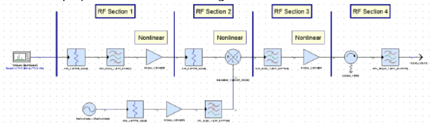
See *Limitations - RF* (users) for additional information.

Theory of Operation

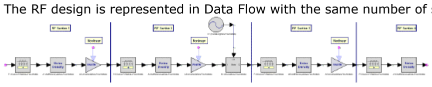
To use a Spectrasys design in a Data Flow schematic (via the RF_Link), an array of frequency domain data is extracted from the Spectrasys design for all paths in the RF design and converted to its time domain representation for use in the Data Flow simulation.

For each path in the RF design non-linear sections are extracted. Each section ends in a non-linearity such as an amplifier or mixer, unless a linear section is the last section in a path. The entire RF design is characterized across a power range from -200 to +60 dBm. The default characterization frequency range is the carrier frequency +/- sample rate / 2. The frequency range characterization extracts the RF circuit frequency response at the carrier frequency which includes all impedance mismatches. Thermal noise is also extracted for each section. Each RF section is modeled in the time domain with Data Flow models that include an FIR filter, additive noise density (if the **Calculate Thermal Noise** option is checked), nonlinear amplifier or mixer. For each mixer encountered the local oscillator frequency, amplitude, and phase are extracted. Any effects of an LO path from the LO source to the LO node of the mixer is accounted for in the LO frequency, amplitude, or phase.

For example, consider this RF design with section boundaries as shown.



The RF design is represented in Data Flow with the same number of sections as shown.



As can be seen from the above figures, the automatic conversion from the frequency domain design to its time domain equivalent is "correct by construction" with proper positioning of linear filtering, additive thermal noise, non-linearities and up or down converting mixers. The time domain equivalent is assured to have time causality. Thus, if a frequency domain characteristic is not time causal, such as a frequency domain characteristic with zero phase shift at all frequencies, it will have an appropriate amount of time delay applied to force it to be causal.

Each RF section is replaced with one or more of these Data Flow models:

- *CustomFIR* (algorithm) used to model the RF small signal gain and phase response versus frequency.
- *AddNDensity* (algorithm) used to model the RF thermal noise versus frequency.
- *Amplifier* (algorithm) used to to model the RF gain and phase change from small signal condition versus power.
- *Mixer* (algorithm) and *Oscillator* (algorithm) used to model frequency conversion.

The Data Flow input to RF_Link must be a *complex envelope signal* (sim) with a non-zero characterization frequency. This signal is typically defined by use of an *Oscillator* (algorithm), *Modulator* (algorithm) or *CxToEnv* (algorithm) model.

When the **Calculate Thermal Noise** option is enabled on the RF Link all noise generated by both passive and active components is extracted across the frequency characterization range and passed to the AddNDenisty block. However, thermal noise from the source or port associated with the input in the RF design is not included.

> **Caution**
> In RF designs normally all RF ports generate thermal noise. However, in data flow simulations thermal noise is not modeled unless the user specifically adds appropriate data flow noise models. The **'Add source thermal noise to input'** option, when checked, will automatically add thermal noise to the data flow input signal driving the RF Link that represents the RF input port noise commonly modeled in an RF design. During an RF Link simulation the RF input port of the RF design does **NOT** generate thermal noise. Is is assumed that input signal to the RF Link contains the correct input thermal noise when the **Calculate Thermal Noise** option has been checked. If the data flow signal driving the RF Link contains noise AND the **'Add source thermal noise to input'** is also checked the total noise may be double counted.

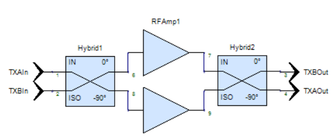
When the **Calculate Phase Noise** option is enabled on the RF Link the phase noise will be extract from the mixer LO's and will be passed to data flow for proper co-simulation.

> **Caution**
> Amplitude noise on the LO is currently being ignored.

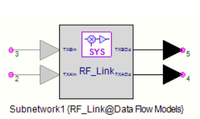
Multiple Input and Output Ports

Gain, phase, and noise is characterized across frequency and power along the RF path from the input to the path output. The previous section illustrates a characterization for a single path from its input to its output. When more than 1 input and 1 output exist then a path characterization of gain, phase, and noise is needed for each input to each output.

For example, for the following dual hybrid matrix amplifier:



The following symbol is created in a dataflow schematic that represents that RF circut:



Internally, 4 characterization paths are created behind the scenes from the each of the inputs to all the outputs.

Path Number	Input	Output
1	TXAin	TxAOut
2	TXAin	TxBOut
3	TXBin	TxAOut
4	TXBin	TxBOut

> **Note**
> The number of characterization paths = **number of inputs** x **number of outputs**. The simulation speed obviously decreases as more paths are needed to characterize the RF system.

See *Simulation - RF* (users) for additional information.

Tutorial

The connection between Data Flow and the RF simulator called Spectrasys is through a part called the *RF Link* (algorithm). An RF Link part can be placed on a data flow schematic in one of two ways.

1. Dragging the RF schematic from the workspace tree and dropping it onto the data flow schematic
2. Using the RF Link part in the part selector or toolbar

Drag and Drop

When dragging and dropping the RF design onto the data flow design the RF Link symbol is automatically configured with the corresponding types and directions of ports used in the RF design.

Part Selector

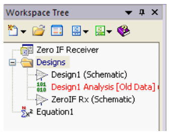
After the RF Link part has been placed on a data flow schematic the specific RF design needs to be selected so that the schematic symbol can be configured correctly. Double click the RF Link symbol and select the desired RF design. The RF Link symbol will automatically configure itself with the corresponding types and directions of ports used in the RF design.

RF / Data Flow Co-Simulation Walk Through

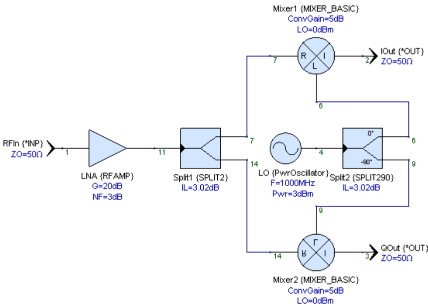
A zero IF receiver will be constructed and simulated in the frequency domain. This receiver will be co-simulated in data flow using QPSK modulation.

Create the Zero IF Receiver

- Create the a new RF schematic and name it: **ZeroIF Rx** (See *Schematics* (users) for additional information)



- Place the RF parts on the schematic



- Change the part parameters to correspond with those shown in the schematic

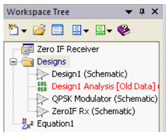
Note
If a System Analysis is to be run on the **ZeroIF Rx** schematic then the input port (RFIn) should be replaced with a Spectrasys **Multisource** part.

Analyze the RF Simulation Results

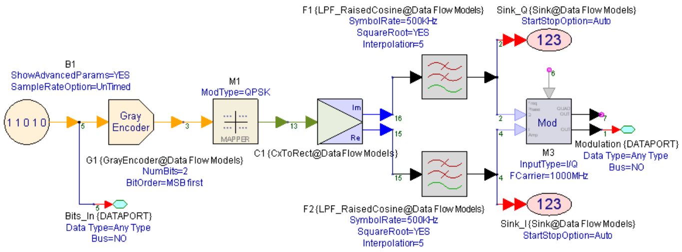
- Add desired Graphs or Tables (See *Graphs* (users) or *Adding a Graph or Table in Spectrasys* (sim) for additional information)

Create the QPSK Modulator

- Create new schematic and name it: **QPSK Modulator**. (See *Schematics* (users) for additional information)



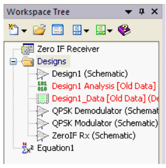
- Place the data flow parts on the schematic



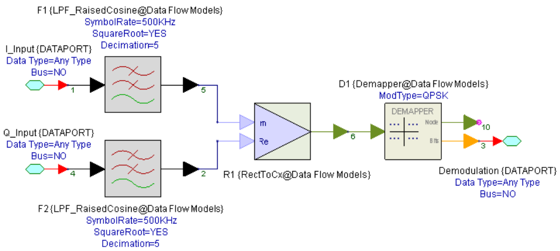
- Change the part parameters to correspond with those shown in the schematic

Create the QPSK Demodulator

- Create new schematic and name it: **QPSK Demodulator**. (See *Schematics* (users) for additional information)



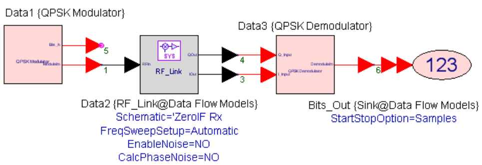
- Place the data flow parts on the schematic



- Change the part parameters to correspond with those shown in the schematic

Create the Co-Simulation Design

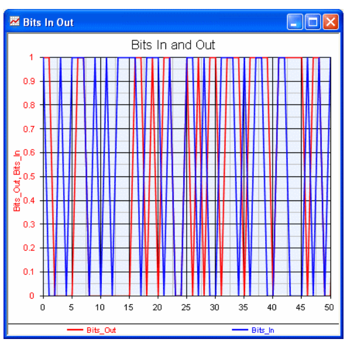
- Drag the **QPSK Modulator icon** from the **Workspace Tree** to **Design1**
- Drag the **ZeroIF Rx icon** from the **Workspace Tree** to **Design1**
- Drag the **QPSK Demodulator icon** from the **Workspace Tree** to **Design1**
- Finish the design by adding a **Sink** as shown



Hint
There are two outputs for the **QPSK Modulator**. Make sure the **Modulation** output is connected to the RF Link.

Analyze the Co-Simulation Results

- The **Data Flow** analysis should already be created and the **Design** parameter should be pointing to the co-simulation schematic named **Design1**
- **Run** the Data Flow analysis (See *Running the Simulation* (sim) for additional information)
- **Add a Graph** and plot **Bits_In** and **Bits_Out**. (See *Graphs* (users) for additional information)



Hint
The **x-axis** on the graph was set to 50 to minimize the amount of data shown.

Schematics

This section describes how to create and use a schematic. A schematic is a graphical way of describing a network of parts connected together through schematic symbols. These parts also contain models that are simulated. The schematic symbols and wiring show the connectivity between models.

Contents

Annotating Schematics

The Annotation button ( ) on the Schematic toolbar gives you access to the Annotation toolbar.

The Annotation toolbar provides tools like lines, circles, and text that you can use to point out details of interest on a schematic, draw a box around a group of components, etc.

> **Hint**
> Double-click a text annotation to set the horizontal and vertical justification (text alignment).

> **Hint**
> **For Advanced users:** An equation can be used for the text. For example, if the workspace contains an equation block with a text variable named **CompanyName**, place **=CompanyName** in the Text field of the title block. The **leading = sign** indicates that the text string is actually an expression. When the title block is drawn, the variable will be evaluated and the result will be displayed in the title block.

Text annotations can display model and parameter info when used within a custom symbol. This is implemented via macro-text-substitution. When symbol text is drawn on a schematic, the displayed text is modified prior to output. For example, Name=%Model% would be displayed as "Name=Resistor" on a symbol using a resistor model. The recognized macro strings are:

1. **%Des%** - Displays the part's designator.
2. **%Model%** - Displays the name of the model attached to the part.
3. **%MODEL%** - Displays the model name in UPPERCASE.
4. **%ParameterName%** - Displays the value of the specified model parameter attached to the part.

Adding Text

Text can be placed directly on a schematic.

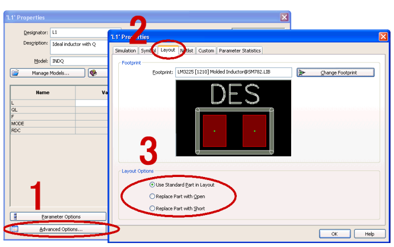
To add text:

1. **Click** the Annotation button ( ) to display the Annotation toolbar.
2. **Click** the **Text** button ( ).
3. **Click** in the Schematic window where you want to place the text.
4. **Type** the **text** into the **Enter 1 or lines of text** field.

Specifying Schematic Part Layout Options

Often in RF circuits, you want to model packaging or component parasitics. You do this by placing lumped parts in series or parallel with the actual part. However, you do not want these parts to display in the layout.

To prevent a schematic part from displaying in a layout:



1. Double-click a part in the schematic and click the Advanced Options button.
2. Click the Layout tab.
3. Click an option. For capacitors, select **Replace Part with Open**. For inductors and resistors, select **Replace Part With Short**.
4. Click **OK**.

Changing the Schematic View

Many times schematics can become so large that the entire schematic is not visible. In these cases panning and zooming helps change the current schematic view.

Panning a Schematic

Panning can be used to move the position of the schematic.

To pan:

- Use the scroll bars to move the page up and down or left and right.
  Or
- Select the Pan Tool (keyboard **P**) from the schematic toolbar. Click and drag the schematic to pan with the mouse.

Zooming a Schematic

Use the zooming features to change the viewing area of the schematic.

Ways to zoom a schematic:

- Click one of the following buttons on the Schematic toolbar:

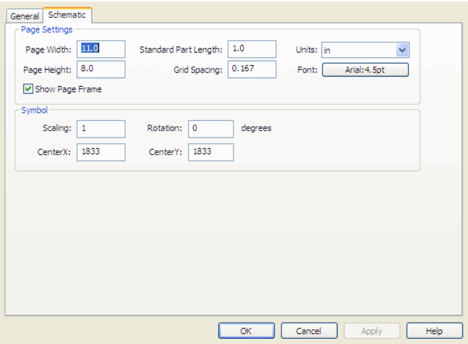
Button	Description	Keyboard	Details
	Zoom an arbitrary area.		Click the schematic and drag the mouse to set the zoom selection rectangle. All items within this rectangle will be zoomed.
	Zoom the schematic to page.	Ctrl+End	Zoom to the page frame.
	Zoom to fit selected parts.		Only selected parts will be zoomed as a group.
	Zoom to fit all schematic objects.	Z	Zoom to include all parts in the schematic.

- Move the **mousewheel** in/out to zoom the schematic in/out
- Use the keyboard **+** and **- keys** to zoom in and out.

Changing Schematic Properties

To change the properties of a schematic:

1. Double-click any empty area of a schematic.
2. Click the **Schematic** tab.
3. Make any changes.
4. Click **OK**.



Page Settings

- **Page Width & Height** - The size of the paper (in current units).
- **Standard Part Length** - The length of a resistor part. Defaults to 1 inch. This setting controls the schematic scaling. (If standard part length is set to 0.5, all parts on the schematic will be half-size.)
- **Grid Spacing** - The distance between grid dots.
- **Units** - The units used by the schematic for its settings.
- **Font** - Default font use when text is placed on the schematic.
- **Show Page Frame** - When checked shows the page outline on the schematic.

Symbol

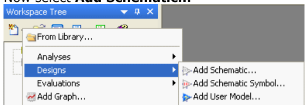
- **Scaling** -
- **Rotation** -
- **CenterX and Y** -

Creating a Simple Schematic

There are two different ways to to create a design in SystemVue. One is the by clicking on the New Item button ( ) on the Workspace Tree toolbar or by right clicking on a folder in the workspace tree.

Method 1 - **Clicking on the New Item Button**

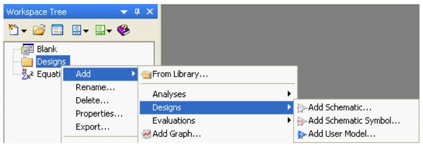
1. Click the New Item button ( ) on the Workspace Tree toolbar
2. Select the **Designs >** submenu
3. Now select **Add Schematic...**



Or

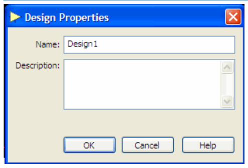
Method 2 - **Right Clicking on a Workspace Folder**

1. Right click on a folder in the workspace tree to bring up the right click menu.
2. Select the **Add >** submenu.
3. Select the **Designs >** submenu
4. Now select **Add Schematic...**

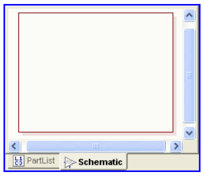


The name of the schematic can then be entered along with an optional description.

> **Note**: The schematic will be added under the folder that was last selected in the workspace tree. (ie. whatever is the "current" folder.) If you want to move it to a different directory simply drag and drop it in the new folder.



A blank schematic will appear.



Manipulating Parts

Connecting Parts

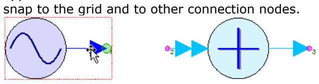
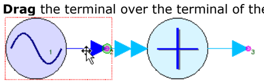
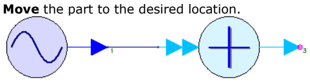
There are three methods that can be used to connect parts together.

Method 1- **Wire Toolbar Buttons:**

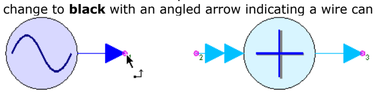
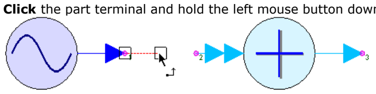
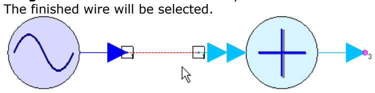
1. **Click** on the **right angle** ( ) or **angled** ( ) toolbar buttons contained on the

schematic toolbar.
2. **Click** and drag to draw the wire on the schematic.

Method 2- **Dragging the Part Terminal:**

1. **Click** the part terminal to be connected. A connection highlight dot (green circle) will appear on the terminal nearest the mouse cursor. This marks the terminal which will snap to the grid and to other connection nodes.

2. **Drag** the terminal over the terminal of the part to be connected.

3. **Move** the part to the desired location.


Method 3- **Dragging a Wire between Terminals:**

1. **Place the mouse** over the part terminal to be connected. The mouse cursor will change to **black** with an angled arrow indicating a wire can be drug from terminal.

2. **Click** the part terminal and hold the left mouse button down will dragging the wire.

3. **Drag** the wire to the terminal of the part to be connected. Release the mouse button. The finished wire will be selected.


For additional information read *Nets, Connection Lines, and Buses* (users) in the Users Guide.

Moving Parts

There is a global option to *keep parts connected* (users) when they are moved or they will be unconnected when they move. The **Alt** key toggles this behavior.

+To move a part:

1. **Click** the part to select it.

> **Hint**
> A small green circle appears on the terminal closest to the mouse. This is the reference terminal for part alignment, connections, and snap points.

2. **Drag** the part to the location of interest.

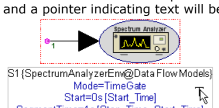
> **Hint**
> Multiple parts can be selected and manipulated at the same time. Click and drag a selection rectangle around the parts of interest in a schematic. Holding down the **Ctrl** key during part selection will select / de-select parts from the group.

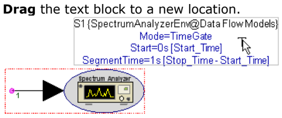
> **Advanced Tip**
> Net names (node numbers) may be re-assigned during a move; when that happens, the first priority is to retain the existing nets attached to ports. Parts which are stationary have the the next highest priority and parts which moved have the lowest priority. This means that if you would like to retain certain existing net/node names (perhaps because you are referencing them in graph measurements), make sure you move the *other* parts of the schamatic, instead of the area you care about.

Moving Part Text

There are three ways to the move the part text.

Method 1- **Drag to Location:**

1. **Click** the part to select it.
2. **Move** the mouse over the part text selection rectangle. The cursor will change to a **T** and a pointer indicating text will be moved on a mouse drag.


3. **Drag** the text block to a new location.


Method 2- **Keyboard Shortcut:**

1. **Click** the part to select it.
2. **Press F4** to rotate through standard text locations of: Top, Bottom, Left, Right, and Center.

Method 3- **Right Mouse Menu:**

1. **Right Click** the part.
2. Select the **Text** menu.
3. Select the desired text location.

Deleting Parts

Parts can be deleted from the schematic.

To delete a part:

1. **Click** the part(s) to be deleted.
2. **Press** the **Del** key

Modifying Part Parameters

To modify part parameters directly on the schematic see *Editing Part Parameters On a Schematic* (users) in the User's Guide.

Placing Parts on a Schematic

Only parts and annotations can be placed on a schematic. Annotation objects are things like text, title blocks, and other drawing objects like polygons, and rectangles. Only parts are connected together through wires or buses.

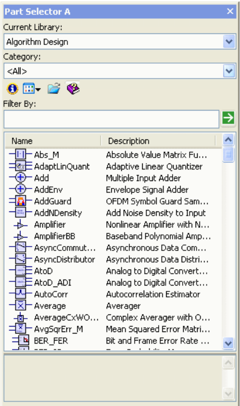
Parts can be placed on the schematic in either of two ways. Through a **part selector** or through **part toolbars**.

> **Hint**
> Some parts can be placed with keyboard short cuts. See *Appendix A Keystroke Commands* (users) for more information.

> **Hint**
> Some parts can be placed by dragging them from the workspace tree to the schematic. When a schematic or sub-network model is dragged in this way a sub-network part is created with an auto-generated schematic symbol. S-parameter datasets can also be placed on the schematic by dragging them.

Method 1 - **Part Selector**

1. Bring up the part selector by clicking on the **Part Selector** button ( ) on the **Schematic** toolbar.



2. **Click** on a part.
3. **Move** the mouse over the schematic. The cursor will change to a plus sign when placed over the schematic.
4. **Click** the schematic where the part is to be placed.

Method 2 - **Part Toolbars**

1. **Click** either the **Data Flow** ( ) or the the **Part Groups** ( ) toolbar on the **Schematic Toolbar** to make visible the desired part toolbar.
2. For example, select the Data Flow toolbar. The toolbar will appear.

3. **Click** the desired part.
4. **Move** the mouse over the schematic. The cursor will change to a plus sign when placed over the schematic.
5. **Click** the schematic where the part is to be placed.

Changing Part Orientation

The user can change the part orientation using keystrokes, a part right click menu, and the Main menu.

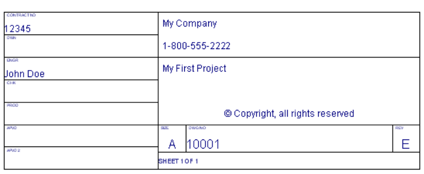
> **Hint**
> When placing the part on the schematic with the mouse the direction of its travel on the left mouse click will determine the initial orientation of the part. For example, if the mouse were being dragged slightly from left to right when the left mouse button is clicked the part would be oriented from left to right on the schematic.

To change the part orientation:

1. **Select** the part by clicking on it. A red selection rectangle will appear around the part.
2. Press **F3** to rotate the part clockwise, **Shift + F3** for counter clockwise, and **F6** for a mirror.
3. Repeat step 2 as necessary.

Title Blocks

A title block is used to document a schematic. It often contains information regarding the name of the schematic, the name of the person who drew it, copyright information, etc. A library of common title blocks ship with the product.



Adding a Title Block

There are two ways to add a title block.

Method 1 - **From the Main Menu:**

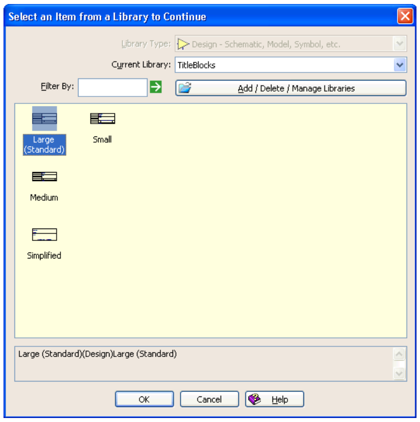
1. **Click** the Schematic menu and select **Add Title Block...**.
2. **Select** the desired title block from the library selector.
3. **Drag** the title block to the location of interest.

Or

Method 2 - **From the Schematic Right Click Menu:**

1. **Right Click** the Schematic and select **Add Title Block...**.
2. **Select** the desired title block from the library selector.
3. **Drag** the title block to the location of interest.

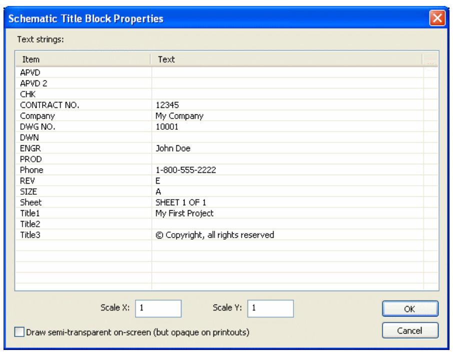
Title Blocks in the Library Selector:



Editing the Title Block

To Edit a Title Block:

1. **Double Click** the title block.
2. **Enter** the desired information.
3. **Click** Ok.



- **Item** - Name or titles of information.

  > **Note**: These titles can only be changed on a custom title block symbol.

- **Text** - The actual text string to be drawn in the title block.
- **Scale X & Y** - Scales the title block. For example, 0.5 is half-size.
- **Draw semi-transparent** - When checked draws a faded title block.

> **Hint**
> **For Advanced users:** An equation can be used for the text. For example, if the workspace contains an equation block with a text variable named **CompanyName**, place **=CompanyName** in the Text field of the title block. The **leading = sign** indicates that the text string is actually an expression. When the title block is drawn, the variable will be evaluated and the result will be displayed in the title block.

Creating a Custom Title Block

The easiest way to create a custom title block is to start with an existing one.

1. Open up the **Library Selector**.
2. Set the **Library Type** to **Design**.
3. Change the **Current Library** to **TitleBlocks**.
4. **Double click** on the title block to be modified.

   > **Note**: This will add the title block as a schematic symbol on the workspace tree.

5. **Edit** title block symbol as needed, using annotations:
   - The text annotation **Name** property will be used as the **Item Name**. The **Show Name** option must be checked in this annotation dialog box to see this name on the schematic.
   - The text annotation **Enter 1 or more lines of text** property will be the text value of the field that appears in the title block.

     > **Hint**
     > For best results, only use 1 line of text and keep it fairly short.

   - **Images**, like a company logo, or any other annotation can be placed on the custom symbol.
6. Save the workspace.
7. On workspace tree, **right-click** the symbol and use **"Copy To"** to place the symbol in a new (or existing) library.
8. To use your new custom title block on a schematic, use **"Add Title Block..."** and select the custom title block from the library it was saved in.

Scripts

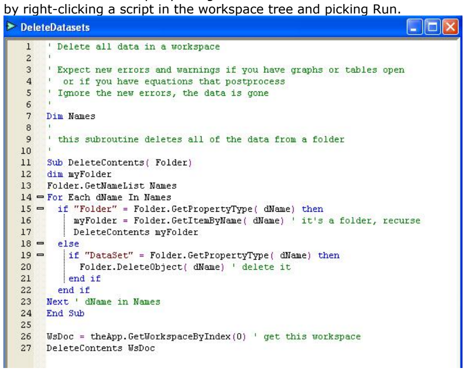
Scripts can be used to perform a variety of functions in SystemVue. Some pre-written script have been included with SystemVue and can be found in the Library Selector by setting the Library Type to "Scripts".

Contents

Adding a Script

To add a script to SystemVue:

- Click the New Item button ( ) on the Workspace Tree toolbar and select **Add Script**.
- Once the script is added, edit script text in the window just like a Notes window.
- The toolbuttons at the top let you run the script or copy the script to the script processor (if you want to edit it there).
- You can also run a script by using an Annotation Button with an embedded script or by right-clicking a script in the workspace tree and picking Run.



Scripts have been color enhanced to improve their readability.

Using Scripts in Programs

Supported Languages: C#, C++, Visual Basic
External Environments: **LabVIEW**™ , **MATLAB**™

A program can be written in any one of the supported languages to communicate with SystemVue using our COM interface. Scripts and commands can be executed in the SystemVue Script Processor from your program. Your program needs to contain the proper COM reference and include the proper header for our COM Interface.

Register the SystemVue COM Interface

In previous releases of Genesys and SystemVue, the COM interface was registered each time the program was run. Starting in the SystemVue 2011.03 and Genesys 2011.06 releases, the COM interface will registered automatically by the installer. If you run an older version of either product, you will need to re-register the COM interface for the program you wish to run.

To register the COM interface, you will need to run the following command in the windows command prompt (Run as administrator):

```
cd C:\Program Files (x86)\SystemVue2011.03\bin
SystemVue.exe /regserver
```

To unregister the COM interface, you will need to run the following command in the windows command prompt (Run as administrator):

```
cd C:\Program Files (x86)\SystemVue2011.03\bin
SystemVue.exe /unregserver
```

Using the COM Interface for SystemVue

1. Add Interop.GENESYS.dll as a COM Reference to your project. Interop.GENESYS.dll is found under Examples\VBScripting\VBBrowser in your SystemVue directory.
2. Import, Use, or Include GENESYS as a header in your program depending on what language you are using.
3. Create an Instance of the GENESYS.Application

Running Scripts from COM Interface

A script can be run from either the RunScript function or the RunScriptFromFile function.

RunScript Function

To use the RunScript function the context of the script you wish to run must be contained in a string variable. The Script Processor works line by line, so the string variable will need to contain a line return character after each line in your script.

> For Example, in VB this is one way you could format a string variable strScript to contain a script that opens a workspace and runs an analysis.
>
> strScript = "OpenWorkspace("C:\Program Files\SystemVue(Version)\Examples\Comms\Bluetooth.wsv")"
> strScript = strScript & vbCrLf & "WsDoc = theApp.GetWorkspaceByIndex(0)"
> strScript = strScript & vbCrLf & "WsDoc.Analyses.DF1.RunAnalysis()"

Once you have formulated a string containing the script that you want to execute within SystemVue, then use the command RunScript to send the script through SystemVue to the script processor. For example, if the GENESYS.Application was instantiated as SystemVueApp and the string containing the script was called strScript:

For VB script
SystemVueApp.RunScript(strScript, ScriptLanguage.genLangVBScript).

For J Script
SystemVueApp.RunScript(strScript, ScriptLanguage.genLangJScript).

RunScriptFromFile Function

An easier method for running a script in SystemVue from your program is to use the RunScriptFromFile function which runs a script from a text file. Simply copy the cotents of a script in SystemVue to a text file and save the file.

> For example, a text file name MyScript.txt contains:
>
> OpenWorkspace("C:\Program Files\SystemVue(Version)\Examples\Comms\Bluetooth.wsv")
> WsDoc = theApp.GetWorkspaceByIndex(0)
> WsDoc.Analyses.DF1.RunAnalysis()

Use the RunScriptFromFile to load the script text file and execute the script. For example, if the GENESYS.Application was instantiated as SystemVueApp and the string containing the path to MyScript.txt was called strPath:

For VB script
SystemVueApp.RunScriptFromFile(strPath, ScriptLanguage.genLangVBScript).

For J Script
SystemVueApp.RunScriptFromFile(strPath, ScriptLanguage.genLangJScript).

Examples

In this section, we will review the COM interface examples that ship with SystemVue. All except the last example in this section preform the following steps, native in each environment:

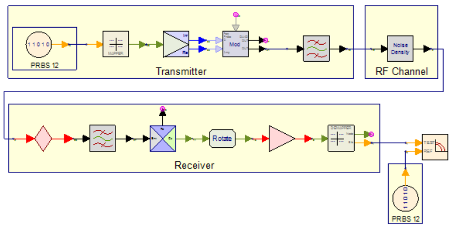
1. Launch SystemVue
2. Open a workspace
3. Sweep a variable
4. Run a simulation
5. Retrieve the result

To simplify use of the COM interface of SystemVue, we have created an example NET DLL component, SystemVueNET.dll, using Visual C#.

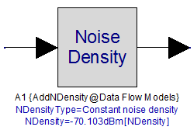
Introduction: SystemVue Eb/N0 Sweep for BER

In this section, we review the workspace used in the the first three COM interface examples. In each of these examples, we will be preforming a bit error rate (BER) analysis by sweeping the Eb/N0 parameter. We can implement this sweep natively in SystemVue using a *parameter sweep* (users). The workspace example is located in "Examples\Comms\BER\QPSK_BER_Coded_Viterbi.wsv". In this workspace, we will be sweeping the Uncoded_QPSK_Design over multiple parameter Eb/N0 values.

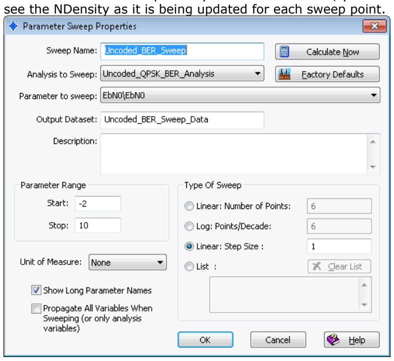
Below is the schematic, note the four distinct sections, transmitter, channel, receiver, and BER measurement:



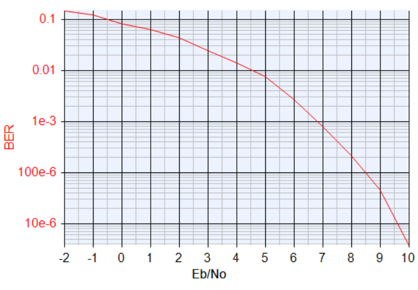
As we preform the BER analysis for a Eb/N0 value, we calculate and modify the value of noise density (NDensity) of the channel:



Below is the parameter sweep in SystemVue, we will be reimplementing this control for the COM interface examples. If you hit calculate now, you can zoom into the channel and see the NDensity as it is being updated for each sweep point.



Finally, after we calculate the Eb/N0 sweep in SystemVue, we can see the BER waterfall plot:



To accomplish this sweep, we first define a equation block declaring that Eb/N0 will be swept:

```
% Eb/No = energy per bit / noise density
% EbN0 is defined in a separate equation block to enable
% updating the variable using external control.  See the
% Automation section in the notes.
EbN0 = ?3
```

In another equation block, we calculate the NDensity using the swept Eb/N0 value:

```
ModPower_dBm = 13 % modulator output power in dBm
SymbolRate = 51.2e+6
ModPower_W = 10^( (ModPower_dBm-30)/10 );
ModPower_Vrms = sqrt( 50*ModPower_W);
ModCarrier = 300e6;
ModAmpSensitivity = ModPower_Vrms*sqrt(2);
SymbolTime = 1/SymbolRate
BitsPerSymbol = 2
% Eb/No = energy per bit / noise density
Eb_dBm = ModPower_dBm - 10*log10( SymbolRate * BitsPerSymbol )
No_dBm = Eb_dBm - EbN0
NDensity = No_dBm
```

Note, since we are using the COM interface, we must declare Eb/N0 in a separate equation block. By doing so, as we change Eb/N0 over COM, the second equation block will be automatically calculated before the simulation is run.

In this example we swept Eb/N0 and displayed the BER results. In the following sections, we will use the SystemVue COM interface to implement the sweep in the following environments:

- *Visual C#*
  - Simplifying the COM Interface using NET DLL component
  - Preforming the BER Analysis
- LabVIEW
- MATLAB

Visual C#

In this example, we use Visual C# to preform the Eb/N0 sweep. The executable is provided at: "Examples\Scripting\C#\QPSK_BER.exe"

When you start it, you will see:

This custom application, enables you to:

- Hit the Run button to preform the sweep
- Hide and unhide the visibility using the check box provided.

To see the sweep in action, unhide SystemVue, zoom into the channel, and watch the NDensity parameter update as each sweep point is evaluated.

The Visual Studio solution is supplied in the "Examples\Scripting\C#\Visual Studio" directory. To customize it, you can use Visual Studio 2008 C# Express Edition (free from Microsoft).

Simplifying the COM Interface using NET DLL component

To help with all of the Eb/N0 examples, we supply an example NET DLL component, named SystemVueNET.dll. This DLL allows us to simplify the management of the COM interface for the QPSK BER examples implemented in C#, LabVIEW, and MATLAB.

In this DLL, we define a class called SystemVue, the file located in "Examples\Scripting\C#\Visual Studio\SystemVueNET\SystemVue.cs":

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using Microsoft.Win32;
namespace SystemVueExample
{
   public class SystemVue
   {
       // Instance of SystemVue application
       GENESYS.Application m_app;
       // Constructor, called when a instance of this class is created
       public SystemVue()
       {
           try
           {
               // Start a new instance of SystemVue
               m_app = new GENESYS.Application();
           }
           catch
           {
               // If we have a exception, the COM server is probably not registered.
               // Register it by running SystemVue.exe /regserver
               m_app = null;
           }
           // By default, make SystemVue hidden
           Visible = false;
       }
       // Member boolean to track visibility
       bool m_bVisible = false;
       // Methods to set/get Visible property of SystemVue
       public bool Visible
       {
           get { return m_bVisible; }
           set
           {
               m_bVisible = value;
               if (m_app != null)
               {
                   m_app.Application.Visible = m_bVisible;
               }
           }
       }
       // Some external environments need a separate method to set visibility
       public void SetVisible(bool bVisible)
       {
           Visible = bVisible;
       }
       // Version number of SystemVue, used to find example area
       static string m_sSystemVueVersion = "2011.03";
       // Return the examples directory path for the version declared above
       public static string ExamplesDirectory()
       {
           RegistryKey hkcu = Registry.CurrentUser;
           string systemVueRegPath;
           systemVueRegPath = "Software\\Agilent\\SystemVue" + m_sSystemVueVersion + "\\System";
           RegistryKey svuRegistry = hkcu.OpenSubKey(systemVueRegPath);
           Object examplesPath = svuRegistry.GetValue("ExamplesPath");
           hkcu.Close();
           return (string)examplesPath;
       }
       // Destructor
       ~SystemVue()
       {
           // Close and save all workspaces
           try
           {
               for (int i = 0; i < m_app.Manager.GetWorkspaceCount(); i++)
               {
                   GENESYS.Workspace workspace = m_app.Manager.GetWorkspaceByIndex(i);
                   // COM interface does not support quitting without saving, so save to temp
file, and then delete it
                   string file = Path.GetTempFileName();
                   workspace.SaveAs(file);
                   File.Delete(file);
               }
           }
           catch
           {
           }
           // Quit the application
           if (m_app != null)
               m_app.Quit();
```

```
        }
        // Run a VB script command
        public bool RunScript(string csScript)
        {
            bool bStatus = true;
            try
            {
                // Run a script, assuming Visual Basic
                m_app.Application.RunScript(csScript, GENESYS.ScriptLanguage.genLangVBScript);
            }
            catch
            {
                bStatus = false;
            }
            return bStatus;
        }
        // Open a workspace, given the path
        public bool OpenWorkspace(string sPath)
        {
            string sCommand;
            sCommand = "OpenWorkspace(\"";
            sCommand += sPath;
            sCommand += "\")";
            return RunScript(sCommand);
        }
        // Set a scalar double parameter
        public bool SetParameter(string sParamPath, double sParamValue)
        {
            bool bSuccess = true;
            bSuccess = RunScript(sParamPath + ".Set( " + sParamValue + " )");
            return bSuccess;
        }
        // Get data from dataset, assuming double
        public double[] GetData(string sDataName)
        {
            GENESYS.IItem item = GetItem(sDataName);
            double[] data = null;
            if (item != null)
            {
                data = (double[])(((GENESYS.IItem)item).GetVarValue(1));
            }
            return data;
        }
        // Find a item in a Genesys item
        public GENESYS.IItem GetItem(string sItemName)
        {
            GENESYS.IItem me = null;
            if (m_app != null)
            {
                me = (GENESYS.IItem)m_app.Manager;
                me = GetItem(me, sItemName);
            }
            return me;
        }
        // Find a item, given a path
        static GENESYS.IItem GetItem(GENESYS.IItem parent, string sItemName)
        {
            GENESYS.IItem item = parent;
            string[] path = sItemName.Split('.');
            try
            {
                foreach (string itemName in path)
                {
                    if (item != null)
                        item = item.GetItemByName(itemName);
                }
            }
            catch
            {
                item = null;
            }
            return item;
        }
    }
}
```

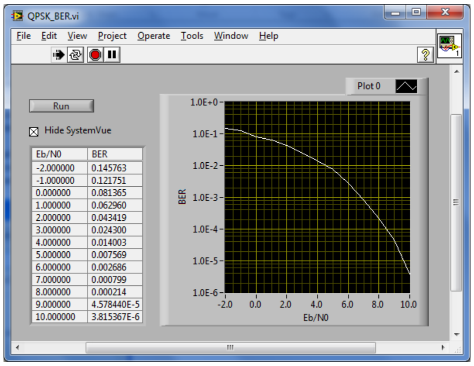
Preforming the BER Analysis

In the Visual Studio solution, the QPSK_BER project defines the GUI and control for the BER sweep. Most of the implementation of this application is in the "Examples\Scripting\C#\Visual Studio\QPSK_BER\QPSK_BER.cs" file. The RunAnalysis method (shown below) preforms the sweep. We use SystemVueNET.dll created in the previous section to interface to the SystemVue COM interface.

```
public void RunAnalysis()
{
   // Create a new instance only if needed
   if (systemVue == null)
   {
       // Start a new instance of SystemVue
       systemVue = new SystemVueExample.SystemVue();
       string workspacePath = SystemVueExample.SystemVue.ExamplesDirectory();
       workspacePath += "\\Comms\\BER\\QPSK_BER_Coded_Viterbi.wsv";
       // Open the workspace
       systemVue.OpenWorkspace(workspacePath);
       systemVue.Visible = Visible;
   }
   // Sweep Eb/N0 -2 to 10 and calculate the BER
   for (int EbN0 = -2; EbN0 <= 10; EbN0++)
   {
       // Set the NDenstity parameter
       systemVue.SetParameter("QPSK_BER_Coded_Viterbi.EbN0.VarBlock.[EbN0]", EbN0);
       // Run the analysis
       systemVue.RunScript(
           "QPSK_BER_Coded_Viterbi.Analyses.Uncoded_QPSK_BER_Analysis.RunAnalysis");
       // Read BER from dataset
       double[] BER = systemVue.GetData(
           "QPSK_BER_Coded_Viterbi.Analyses.Uncoded_QPSK_BER_Data.Eqns.VarBlock.B11_BER");
       // BER could be null if user manually exited SystemVue
       if (BER != null)
       {
           QPSK_BER.SimulationResult newSim = new QPSK_BER.SimulationResult();
           newSim.BER = BER[0];
           newSim.EbN0 = EbN0;
           newSim.Test = BER[0] > .1 ? "Fail" : "Pass";
           m_SimulationResults.Add(newSim);
       }
  }
}
```

LabVIEW

In this example, we use LabVIEW to implement the BER analysis. The compiled executable is available in "Examples\Scripting\LabVIEW\QPSK_BER_vi.exe":



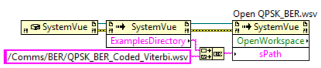
To run it, you will need to install "LabVIEW Run-Time Engine 2009 - (32-bit Minimum RTE)" available free from National Instruments at:
http://joule.ni.com/nidu/cds/view/p/id/1406/lang/en

As in the previous example, we use SystemVueNET.dll to manage the SystemVue COM interface.

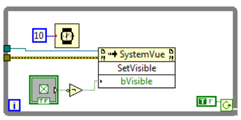
The LabVIEW vi is defined in the "Examples\Scripting\LabVIEW\QPSK_BER.vi" file. You will need LabVIEW 2009 or later to open the vi file.

The LabVIEW application provides the implementation for:

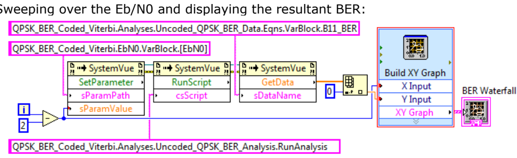
- Starting SystemVue and loading the workspace:



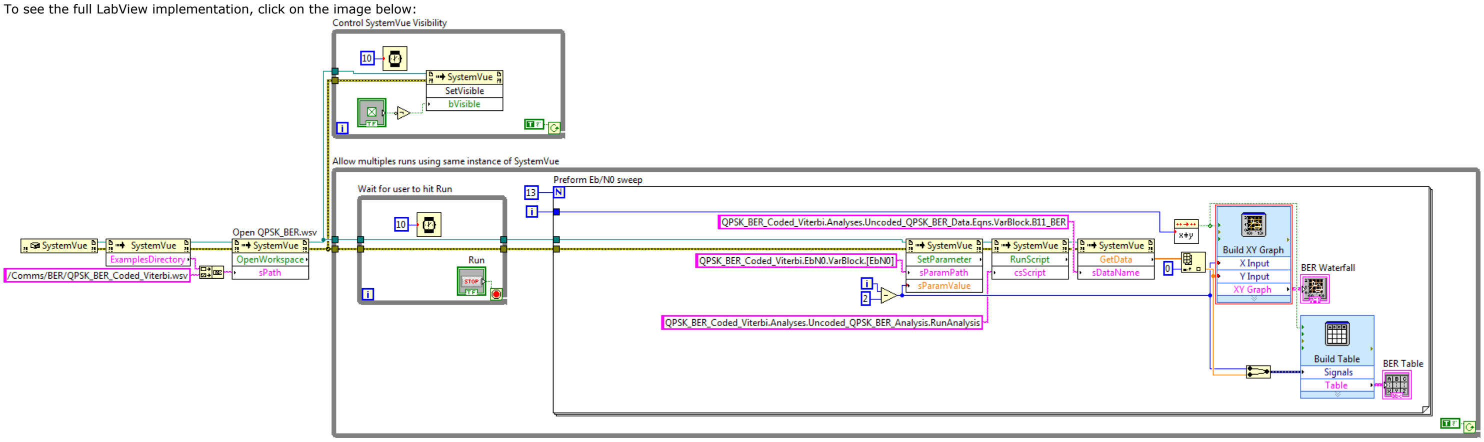
- Toggling the visibility of SystemVue:



- Sweeping over the Eb/N0 and displaying the resultant BER:



To see the full LabView implementation, click on the image below:



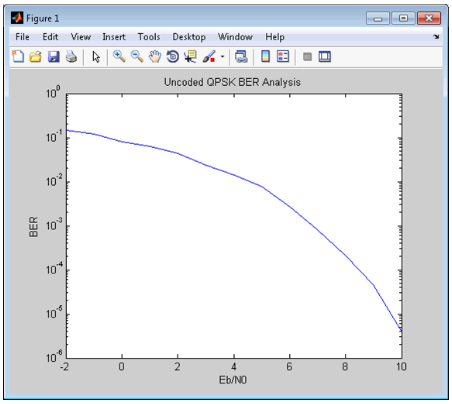
MATLAB

In this example, we use MATLAB to implement the BER analysis. The MATLAB script is defined in the "Examples\Scripting\MATLAB\QPSK_BER.m" file.

As in the previous example, we use SystemVueNET.dll created above to interface to the SystemVue COM interface.

```
% Find the directory path where this file is located
pathToDLL = fileparts( mfilename('fullpath'));
% Load the assembly in this directory (source code in C# example area)
%NET.addAssembly([pathToDLL '/SystemVueNET.dll']);
% Open SystemVue and the workspace that we are interested in
if exist( 'systemVue') == false
   % Start a new instance of SystemVue
   systemVue = SystemVueExample.SystemVue();
   % Hide SystemVue
   systemVue.Visible = false;
   % Get the examples directory path
   examplesDirectory = char(systemVue.ExamplesDirectory());
   % Define workspace path to build directory examples directory
   workspacePath = [examplesDirectory '/Comms/BER/QPSK_BER_Coded_Viterbi.wsv'];
   % Open the workspace
   systemVue.OpenWorkspace(workspacePath);
end
% Index into results matrix
i = 1;
% Sweep Eb/N0 -2 to 10 and calculate the BER
for j =-2:10,
   % Set EbN0
   EbN0(i) = j;
   systemVue.SetParameter('QPSK_BER_Coded_Viterbi.EbN0.VarBlock.[EbN0]', EbN0(i));
   % Run the analysis
   systemVue.RunScript('QPSK_BER_Coded_Viterbi.Analyses.Uncoded_QPSK_BER_Analysis.RunAnalysis');
   % Read BER from dataset
   data = systemVue.GetData('QPSK_BER_Coded_Viterbi.Analyses.Uncoded_QPSK_BER_Data.Eqns.VarBlock.B
11_BER');
```

```
   BER(i) = data(1);
   % Display NDensity and BER on the console window
   disp(['Eb/N0 = ' num2str(EbN0(i)), ' BER = ', num2str(BER(i))]);
   % Increment index into results matrix
   i = i+1;
end
%plot the results
semilogy(EbN0,BER)
xlabel('Eb/N0')
ylabel('BER')
title('Uncoded QPSK BER Analysis')
```



Visual Basic

The VBBrowser communicates to SystemVue through the COM interface. The source code for the VBBrowser is located in "Examples\Scripting\Visual Basic\Browser\MainForm.vb" for your viewing.

The executable is available in "Examples\Scripting\Visual Basic\Browser\VBBrowser.exe". This application will let you explore an opened workspace, with it you can find the path to the items in your workspace to use in your automation scripts. To learn more about this application, refer to the *Example Exploring the Workspace Using Visual Basic* (users) documentation.

Creating Script Objects

In SystemVue, all designs and their components are objects that you can refer to by name. In the following example, there is a design named Design1 and it has a SineGen source called S1

Typically, most scripts start out by defining a variable to be the workspace object. In the example below the workspace object was defined by WsDoc = theApp.GetWorkspaceByIndex(0)

To create a sample script object:

1. Set the frequency of a SineGen source named S1 to 12000 Hz
   WsDoc.Design1.PartList.S1.ParamSet.Frequency.Set(12000)
2. Define an object pointing to the S1 part parameter set.
   MySub=WsDoc.Design1.PartList.S1.ParamSet
3. Set the frequency parameter to 12000 Hz
   MySub.Frequency.Set(12000)
4. Set the amplitude parameter to 2 V
   MySub.Amplitude.Set(2)

Set uses the parameter's defined unit of measure.

There is an object browser example using Visual Basic in the SystemVue Examples\VBBrowser directory. This example shows you how to:

- Connect to SystemVue from Visual Basic.
- Browse objects in SystemVue.
- Execute any method in SystemVue.

See the doc on VBBrowser for more details.

Example: Exploring the Workspace Using Visual Basic

VBBrowser

The VBBrowser communicates to SystemVue through the COM interface. The source code for the VBBrowser is located in "Examples\Scripting\Visual Basic\Browser\MainForm.vb" for your viewing.

The executable is available in "Examples\Scripting\Visual Basic\Browser\VBBrowser.exe". This application will let you explore an opened workspace, with it you can find the path to the items in your workspace to use in your automation scripts. To learn more about this application, refer to the *Example Exploring the Workspace Using Visual Basic* (users) documentation.

(SystemVue Browser)

The VBBrowser is used to browse objects in SystemVue. This is an interactive program that allows a user to see what functions are available to call within the script processor. The program communicates with one active instance of the SystemVue program. The browser looks at the current workspace and retrieves objects and items from it.

Running the VBBrowser

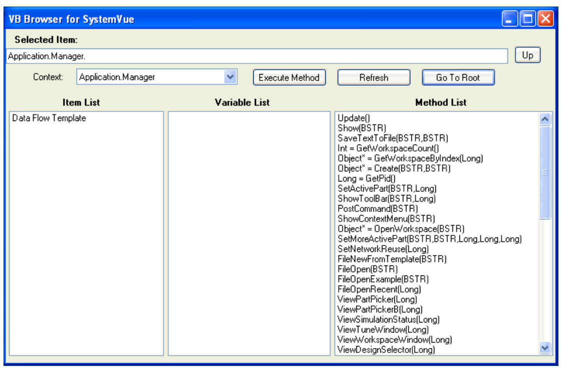
The VBBrowser is located in the Examples\VBScripting\VBBrowser folder of your SystemVue directory. Source code for the VBBrowser can be found in this same folder in the file called MainForm.vb. The files Interop.GENESYS.dll and VBBrowser.exe were created following the instructions found in the ReadMe.txt located in the same folder.

There are two ways to launch the VBBrowser

1. Run the VBBrowser while you have a SystemVue Running.
2. Launch the VBBrower without SystemVue Running. The VBBrowser will launch as well as SystemVue.

> If you load another workspace in SystemVue while the VBBrowser is running it is best to click the Go To Root button to avoid errors. Clicking the Refresh or Up button will throw an error and then load the root.

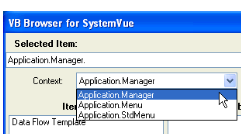
Contents of the VBBrowser



General

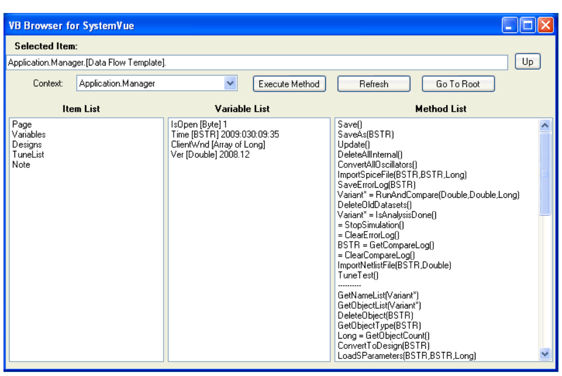
The Selected Item box contains the syntax for the script that you can execute by clicking the Execute Method button.

The Context drop box contains three items



1. Application.Manager (default) Sets the Item List to the context of the workspace tree.
2. Application.Menu Sets the Item List to the context of the current Menu Bar in SystemVue
3. Application.StdMenu - Sets the Item List to the context of the standard Menu Bar in SystemVue

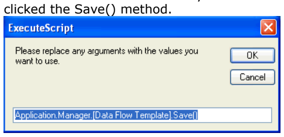
Lists



Item List - The window contains a list of all the items found in the current context. If nothing appears in the window you can click the Refresh button to refresh the context. Clicking on an item in this list will show you a list of sub items. Note that the sub items correspond to the items inside the opened workspace. Notice that as you click items, the text in the selected item box changes. The first thing you should see (in the default context) in the Item list is the name of the workspace(s) that are loaded in SystemVue. In the example above you would see Data Flow Template as the first item in the list.

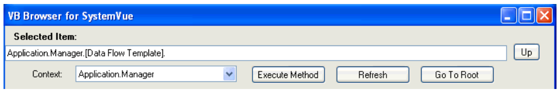
Variable List - The window contains a list of properties, variables, or parameters that are associated with the current item. Items in this list can be called as a property to an item.

Method List - The window contains a list of the methods that can be used with the current item. Notice that by double clicking on a method the ExecuteScript window pops up with current syntax of the method youve selected. This syntax is generated from the Selected Item text box and the method you have clicked. This is what would pop up if you double clicked the Save() method.



You can execute this one line script by clicking on OK. A script processor window will not pop up in SystemVue, so you may not always know if it worked or not. If you need to execute many lines it is suggested to use a script. The ExecuteScript window is best used as a guide to get the correct syntax for writing your own script.

Buttons



Up - The button sets the Item List to the parent item of the current Item List window

Execute Method The button will bring up the ExecuteScript window that shows the syntax for the current Selected Item and gives the option to run it or not.

Refresh - The button reloads the items in the three lists.

Go To Root - The button sets the Item List to the top most parent.

Example Running a BER Analysis Controlled From LabVIEW, MATLAB, or C#

In this section, we will review the COM interface examples that ship with SystemVue. All except the last example in this section preform the following steps, native in each environment:

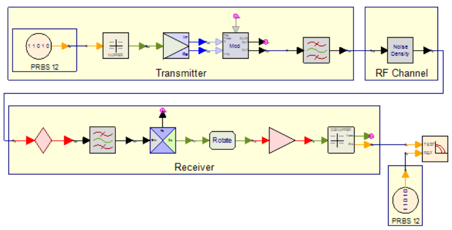
1. Launch SystemVue
2. Open a workspace
3. Sweep a variable
4. Run a simulation
5. Retrieve the result

To simplify use of the COM interface of SystemVue, we have created an example NET DLL component, SystemVueNET.dll.

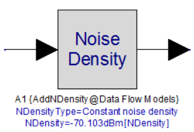
Introduction: SystemVue Eb/N0 Sweep for BER

In this section, we review the workspace used in the the first three COM interface examples. In each of these examples, we will be preforming a bit error rate (BER) analysis by sweeping the Eb/N0 parameter. We can implement this sweep natively in SystemVue using a *parameter sweep* (users). The workspace example is located in "Examples\Comms\BER\QPSK_BER_Coded_Viterbi.wsv". In this workspace, we will be sweeping the Uncoded_QPSK_Design over multiple parameter Eb/N0 values.

Below is the schematic, note the four distinct sections, transmitter, channel, receiver, and BER measurement:

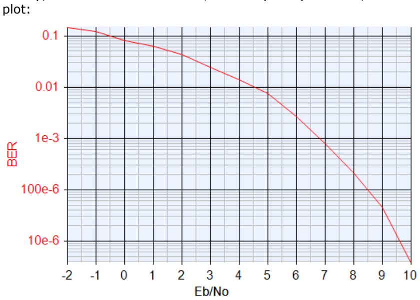


As we preform the BER analysis for a Eb/N0 value, we calculate and modify the value of noise density (NDensity) of the channel:



Below is the parameter sweep in SystemVue, we will be reimplementing this control for the COM interface examples. If you hit calculate now, you can zoom into the channel and see the NDensity as it is being updated for each sweep point.

Finally, after we calculate the Eb/N0 sweep in SystemVue, we can see the BER waterfall plot:



To accomplish this sweep, we first define a equation block declaring that Eb/N0 will be swept:

```
% Eb/No = energy per bit / noise density
% EbN0 is defined in a separate equation block to enable
% updating the variable using external control.  See the
% Automation section in the notes.
EbN0 = ?3
```

In another equation block, we calculate the NDensity using the swept Eb/N0 value:

```
ModPower_dBm = 13 % modulator output power in dBm
SymbolRate = 51.2e+6
ModPower_W = 10^( (ModPower_dBm-30)/10 );
ModPower_Vrms = sqrt( 50*ModPower_W);
ModCarrier = 300e6;
ModAmpSensitivity = ModPower_Vrms*sqrt(2);
SymbolTime = 1/SymbolRate
BitsPerSymbol = 2
% Eb/No = energy per bit / noise density
Eb_dBm = ModPower_dBm - 10*log10( SymbolRate * BitsPerSymbol )
No_dBm = Eb_dBm - EbN0
NDensity = No_dBm
```

Note, since we are using the COM interface, we must declare Eb/N0 in a separate equation block. By doing so, as we change Eb/N0 over COM, the second equation block will be automatically calculated before the simulation is run.

In this example we swept Eb/N0 and displayed the BER results. In the following sections, we will use the SystemVue COM interface to implement the sweep in the following environments:

- Visual C#
  - Simplifying the COM Interface using NET DLL component
  - Preforming the BER Analysis
- LabVIEW
- MATLAB

Visual C#

In this example, we use Visual C# to preform the Eb/N0 sweep. The executable is provided at: "Examples\Scripting\C#\QPSK_BER.exe"

When you start it, you will see:

This custom application, enables you to:

- Hit the Run button to preform the sweep
- Hide and unhide the visibility using the check box provided.

To see the sweep in action, unhide SystemVue, zoom into the channel, and watch the NDensity parameter update as each sweep point is evaluated.

The Visual Studio solution is supplied in the "Examples\Scripting\C#\Visual Studio" directory. To customize it, you can use Visual Studio 2008 C# Express Edition (free from Microsoft).

Simplifying the COM Interface using NET DLL component

To help with all of the Eb/N0 examples, we supply an example NET DLL component, named SystemVueNET.dll. This DLL allows us to simplify the management of the COM interface for the QPSK BER examples implemented in *C#*, *LabVIEW*, and *MATLAB*.

In this DLL, we define a class called SystemVue, the file located in "Examples\Scripting\C#\Visual Studio\SystemVueNET\SystemVue.cs":

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using Microsoft.Win32;
namespace SystemVueExample
{
   public class SystemVue
   {
       // Instance of SystemVue application
       GENESYS.Application m_app;
       // Constructor, called when a instance of this class is created
       public SystemVue()
       {
           try
           {
               // Start a new instance of SystemVue
               m_app = new GENESYS.Application();
           }
           catch
           {
               // If we have a exception, the COM server is probably not registered.
               // Register it by running SystemVue.exe /regserver
               m_app = null;
           }
           // By default, make SystemVue hidden
           Visible = false;
       }
       // Member boolean to track visibility
       bool m_bVisible = false;
       // Methods to set/get Visible property of SystemVue
       public bool Visible
       {
           get { return m_bVisible; }
           set
           {
               m_bVisible = value;
               if (m_app != null)
               {
                   m_app.Application.Visible = m_bVisible;
               }
           }
       }
       // Some external environments need a separate method to set visibility
       public void SetVisible(bool bVisible)
       {
           Visible = bVisible;
       }
       // Version number of SystemVue, used to find example area
       static string m_sSystemVueVersion = "2011.03";
       // Return the examples directory path for the version declared above
       public static string ExamplesDirectory()
       {
           RegistryKey hkcu = Registry.CurrentUser;
           string systemVueRegPath;
           systemVueRegPath = "Software\\Agilent\\SystemVue" + m_sSystemVueVersion + "\\System";
           RegistryKey svuRegistry = hkcu.OpenSubKey(systemVueRegPath);
           Object examplesPath = svuRegistry.GetValue("ExamplesPath");
           hkcu.Close();
           return (string)examplesPath;
       }
       // Destructor
       ~SystemVue()
       {
           // Close and save all workspaces
           try
           {
               for (int i = 0; i < m_app.Manager.GetWorkspaceCount(); i++)
               {
                   GENESYS.Workspace workspace = m_app.Manager.GetWorkspaceByIndex(i);
                   // COM interface does not support quitting without saving, so save to temp
file, and then delete it
                   string file = Path.GetTempFileName();
                   workspace.SaveAs(file);
                   File.Delete(file);
               }
           }
           catch
           {
           }
           // Quit the application
           if (m_app != null)
               m_app.Quit();
       }
       // Run a VB script command
       public bool RunScript(string csScript)
       {
           bool bStatus = true;
           try
           {
               // Run a script, assuming Visual Basic
               m_app.Application.RunScript(csScript, GENESYS.ScriptLanguage.genLangVBScript);
           }
           catch
           {
               bStatus = false;
           }
           return bStatus;
       }
       // Open a workspace, given the path
       public bool OpenWorkspace(string sPath)
       {
           string sCommand;
           sCommand = "OpenWorkspace(\"";
           sCommand += sPath;
           sCommand += "\")";
           return RunScript(sCommand);
       }
       // Set a scalar double parameter
       public bool SetParameter(string sParamPath, double sParamValue)
       {
           bool bSuccess = true;
           bSuccess = RunScript(sParamPath + ".Set( " + sParamValue + " )");
           return bSuccess;
       }
       // Get data from dataset, assuming double
       public double[] GetData(string sDataName)
       {
           GENESYS.IItem item = GetItem(sDataName);
           double[] data = null;
           if (item != null)
           {
               data = (double[])(((GENESYS.IItem)item).GetVarValue(1));
           }
```

```
            return data;
        }
        // Find a item in a Genesys item
        public GENESYS.IItem GetItem(string sItemName)
        {
            GENESYS.IItem me = null;
            if (m_app != null)
            {
                me = (GENESYS.IItem)m_app.Manager;
                me = GetItem(me, sItemName);
            }
            return me;
        }
        // Find a item, given a path
        static GENESYS.IItem GetItem(GENESYS.IItem parent, string sItemName)
        {
            GENESYS.IItem item = parent;
            string[] path = sItemName.Split('.');
            try
            {
                foreach (string itemName in path)
                {
                    if (item != null)
                        item = item.GetItemByName(itemName);
                }
            }
            catch
            {
                item = null;
            }
            return item;
        }
    }
}
```

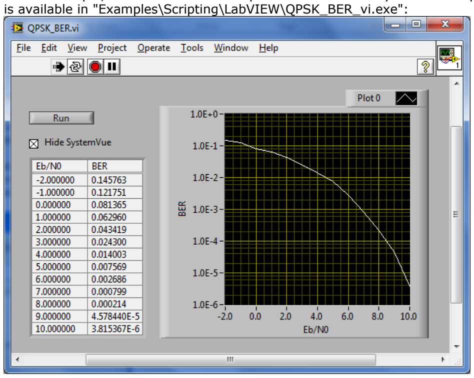
Preforming the BER Analysis

In the Visual Studio solution, the QPSK_BER project defines the GUI and control for the BER sweep. Most of the implementation of this application is in the "Examples\Scripting\C#\Visual Studio\QPSK_BER\QPSK_BER.cs" file. The RunAnalysis method (shown below) preforms the sweep. We use SystemVueNET.dll created in the previous section to interface to the SystemVue COM interface.

```
public void RunAnalysis()
{
   // Create a new instance only if needed
   if (systemVue == null)
   {
       // Start a new instance of SystemVue
       systemVue = new SystemVueExample.SystemVue();
       string workspacePath = SystemVueExample.SystemVue.ExamplesDirectory();
       workspacePath += "\\Comms\\BER\\QPSK_BER_Coded_Viterbi.wsv";
       // Open the workspace
       systemVue.OpenWorkspace(workspacePath);
       systemVue.Visible = Visible;
   }
   // Sweep Eb/N0 -2 to 10 and calculate the BER
   for (int EbN0 = -2; EbN0 <= 10; EbN0++)
   {
       // Set the NDenstity parameter
       systemVue.SetParameter("QPSK_BER_Coded_Viterbi.EbN0.VarBlock.[EbN0]", EbN0);
       // Run the analysis
       systemVue.RunScript(
           "QPSK_BER_Coded_Viterbi.Analyses.Uncoded_QPSK_BER_Analysis.RunAnalysis");
       // Read BER from dataset
       double[] BER = systemVue.GetData(
           "QPSK_BER_Coded_Viterbi.Analyses.Uncoded_QPSK_BER_Data.Eqns.VarBlock.B11_BER");
       // BER could be null if user manually exited SystemVue
       if (BER != null)
       {
           QPSK_BER.SimulationResult newSim = new QPSK_BER.SimulationResult();
           newSim.BER = BER[0];
           newSim.EbN0 = EbN0;
           newSim.Test = BER[0] > .1 ? "Fail" : "Pass";
           m_SimulationResults.Add(newSim);
       }
  }
}
```

LabVIEW

In this example, we use LabVIEW to implement the BER analysis. The compiled executable is available in "Examples\Scripting\LabVIEW\QPSK_BER_vi.exe":



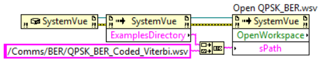
To run it, you will need to install "LabVIEW Run-Time Engine 2009 - (32-bit Minimum RTE)" available free from National Instruments at: http://joule.ni.com/nidu/cds/view/p/id/1406/lang/en

As in the previous example, we use SystemVueNET.dll to manage the SystemVue COM interface.

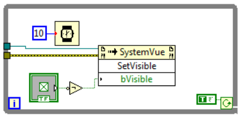
The LabVIEW vi is defined in the "Examples\Scripting\LabVIEW\QPSK_BER.vi" file. You will need LabVIEW 2009 or later to open the vi file.

The LabVIEW application provides the implementation for:

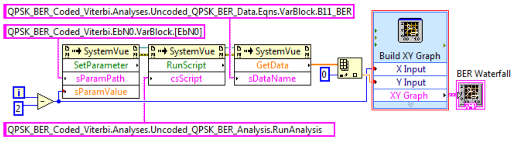
- Starting SystemVue and loading the workspace:



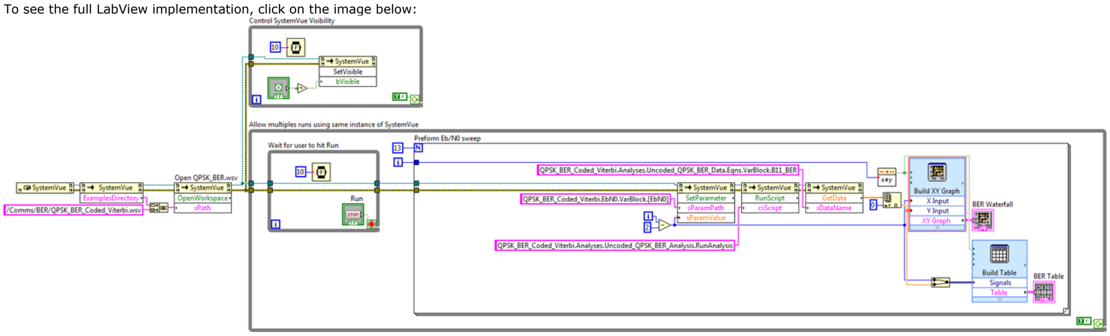
- Toggling the visibility of SystemVue:



- Sweeping over the Eb/N0 and displaying the resultant BER:

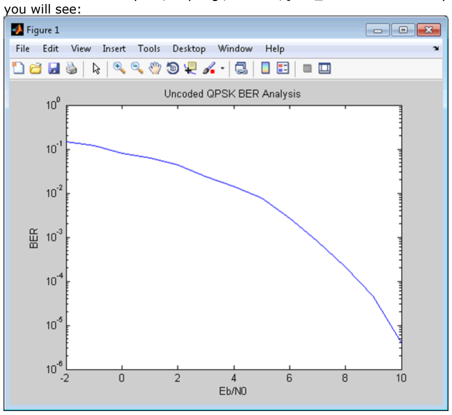


To see the full LabView implementation, click on the image below:



MATLAB

In this example, we use MATLAB to implement the BER analysis. The MATLAB script is defined in the "Examples\Scripting\MATLAB\QPSK_BER.m" file. When you run the script, you will see:



As in the previous example, we use SystemVueNET.dll created above to interface to the SystemVue COM interface.

```
% Find the directory path where this file is located
pathToDLL = fileparts( mfilename('fullpath'));
% Load the assembly in this directory (source code in C# example area)
%NET.addAssembly([pathToDLL '/SystemVueNET.dll']);
% Open SystemVue and the workspace that we are interested in
if exist( 'systemVue') == false
   % Start a new instance of SystemVue
   systemVue = SystemVueExample.SystemVue();
   % Hide SystemVue
   systemVue.Visible = false;
   % Get the examples directory path
   examplesDirectory = char(systemVue.ExamplesDirectory());
   % Define workspace path to build directory examples directory
   workspacePath = [examplesDirectory '/Comms/BER/QPSK_BER_Coded_Viterbi.wsv'];
   % Open the workspace
   systemVue.OpenWorkspace(workspacePath);
end
% Index into results matrix
i = 1;
% Sweep Eb/N0 -2 to 10 and calculate the BER
for j =-2:10,
   % Set EbN0
   EbN0(i) = j;
   systemVue.SetParameter('QPSK_BER_Coded_Viterbi.EbN0.VarBlock.[EbN0]', EbN0(i));
   % Run the analysis
   systemVue.RunScript('QPSK_BER_Coded_Viterbi.Analyses.Uncoded_QPSK_BER_Analysis.RunAnalysis');
   % Read BER from dataset
   data = systemVue.GetData('QPSK_BER_Coded_Viterbi.Analyses.Uncoded_QPSK_BER_Data.Eqns.VarBlock.B
11_BER');
   BER(i) = data(1);
   % Display NDensity and BER on the console window
   disp(['Eb/N0 = ' num2str(EbN0(i)), ' BER = ', num2str(BER(i))]);
   % Increment index into results matrix
   i = i+1;
end
%plot the results
semilogy(EbN0,BER)
xlabel('Eb/N0')
ylabel('BER')
title('Uncoded QPSK BER Analysis')
```

Example: Running a Script from Microsoft Excel

Microsoft Excel has a VB Script engine that one can use to script other applications that support a COM interface. In the case of SystemVue, this means that a Script can be written in Microsoft Excel that opens SystemVue, does something such as load a workspace and run simulations, collects data, and processes the data. For information on accessing the VBScript development editor in Microsoft Excel, see your version of Excel's Help.

The global Windows name for SystemVue's COM server is GENESYS. When SystemVue runs, it registers itself with the Windows operating system by name so that a script can access it (including run an instance of it).

The first thing one must do to be able to access the SystemVue COM server in Excel is to make it visible to Excel by setting it as a "Reference". In the Microsoft Visual Basic editor in Excel, you must declare "GENESYS" as a reference, and this is normally done by accessing the References dialog box via "Tools/References... "

Note: GENESYS will only appear in the References list only if SystemVue has been installed and run at least once.

Now, the SystemVue COM server can be accessed in a VBScript module by the name "GENESYS". Create a new VBScript module by right-clicking on your VBA Project in the

Project explorer and selecting "Insert... / Module".

The following code snippet shows the simplest possible script which simply opens an instance of SystemVue:

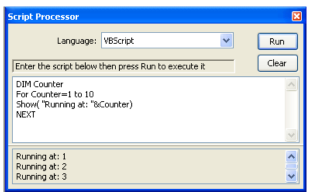
```
Sub myScript()
 Dim comServer As GENESYS.Application  ' Declare variable that references our COM server
 Set comServer = CreateObject("Genesys.Application")   ' Open an instance of the application
End Sub
```

For illustrative purposes, here is a more involved VBScript which opens SystemVue, opens a workspace named "MyWorkspace.wsx", Runs a particular analysis that is located in the workspace, gets data from the dataset, and sets the data into an excel spreadsheet:

```
Sub myScript()
   Dim oGen As GENESYS.Application
   Dim WsDoc As GENESYS.Workspace
   Dim Stri As String

   ' Open Genesys
   Set oGen = CreateObject("Genesys.Application")
   ' Load workspace
   oGen.Manager.OpenWorkspace ("C:\Workspaces\MyWorkspace.wsx")
   ' Get Workspace
   Set WsDoc = oGen.Manager.GetWorkspaceByIndex(0)
   ' Run Analysis called Analysis1
   WsDoc.Designs.Analysis1.RunAnalysis
   ' Get V1 variable from Dataset named MyData
   arr = WsDoc.Designs.MyData.Eqns.VarBlock.V1.GetValue()
   ' Save the workspace
   oGen.Menu.File.Save.Execute
   ' Exit(optional)
   oGen.Menu.File.Exit.Execute
   Dim oXL As Excel.Application
   Dim oWB As Excel.Workbook
   Dim oSheet As Excel.Worksheet
   Dim oRng As Excel.Range
   Dim iNumQtrs As Integer

   Set oXL = Excel.Application  ' Activate Excel
   oXL.Visible = True
   ' Set active Workbook
   Set oWB = oXL.Workbooks.Application.ActiveWorkbook
   ' Set active Sheet
   Set oSheet = oWB.ActiveSheet

   ' output first 201 datapoints to excel
   For i = 0 To 200
       oSheet.Cells(i + 1, 1).Value = arr(i)
   Next i

   Exit Sub
Err_Handler:
     MsgBox Err.Description, vbCritical, "Error: " & Err.Number
End Sub
```

Script Processor

A script contains objects that let you control SystemVue using industry-standard scripting languages. Scripts specifically control SystemVue operations and are very different from equations, which relate variables in SystemVue. Use scripts to load files, save files, save data sets, and change object parameters. Create and run scripts using the Script Processor window. Add the scripts to SystemVue or to a specific design.



SystemVue supports scripts written in both VBScript and JScript. These are standard programming languages not written by Agilent. Documentation for VBScript and JScript is widely available on the Web.

> Note: The latest version of scripting allows scripting from Visual Basic or C__, access to all SystemVue menu items, customization of menus, and custom optimization. For more information on using any of these features, please contact Agilent directly or check the latest Help files at Agilent EEsof EDA Documentation .

To run a script:

1. Click **Tools** on the SystemVue menu and select **Script Processor**.
2. Type or copy a script in the box.
3. Click the **Run** button.

To add a script to SystemVue:

1. Click the New Item button ( ) on the Workspace Tree toolbar and select **Add Script**.
2. Once the script is added, edit script text in the window just like a Notes window.

Script Verbs

Some SystemVue objects have verbs you can use, including a few global verbs that are applicable to the program.

This table contains a list of all the available functions to use in scripts. The functions are organized by what type of object or item they can be called on. For example, functions in the Dataset table can be used off of datasets in your workspace. The VBBrowser is also very helpful in showing what functions can be used on what objects.

The examples in the table below were created using the Data Flow Template.wsv as the opened workspace. The Data Flow Template.wsv workspace can be found in the Template folder of the SystemVue directory. These examples work with the Data Flow Template.wsv example, but can be applied to any workspace. To see the return value of any function in the script processor you can use the Show() function. If the return value is an object it may be necessary to use the GetName property before you can use the Show() function to display the name of the object in the script processor. By default, any file created by a function call is created in the same directory as the workspace currently opened unless a path is specified. Any function that takes a file name as a string parameter can also take a string containing the path of a file as a parameter.

Some sample scripts have been included with SystemVue and can be used as a reference in writing your own. These scripts are located in the Library Selector under the Library Type "Script".

> For all the examples below w = Application.Manager.GetWorkspaceByIndex(0). This sets the variable "w" to the current workspace.

All Main SystemVue Objects

Syntax	Description	Example
ExportToLibrary(bstr LibName)	Export the object to a library.	w.ExportToLibrary("Test")
ImportFromLibrary(bstr LibType, bstr LibName, bstr PartName)	Import an object from the library.	w.ImportFromLibrary("Dataset", "Test", "myData")
GetLibrary(bstr LibType, bstr LibName)	Get the library specified by its type and its name. Use the Library Selector as a reference for the inputs.	result = w.GetLibrary("Design", "SymbolsQtr")
GetRegisteredModels	Gets a list of the registered models	w.GetRegisteredModels result
ImportFromLibrary(bstr LibType, bstr LibName, bstr PartName)	Import a part from a library. The parameters are the library type, the library name, and the name of the part. Use the Library Selector in SystemVue as a reference to find all these inputs.	w.ImportFromLibrary "Design", "Symbols", "OSC"
OpenWindow	Open a view of the object.	w.Designs.Signal.OpenWindow()
CloseWindow	Close any open views.	w.Designs.Signal.CloseWindow()
SelectAll	Select all 'parts' in the object	w.Designs.Design1.SelectAll
SelectNone	Deselect all 'parts' in the object	w.Designs.Design1.SelectNone

Analysis

Syntax	Description	Example
ClearModelCache()	Clear the model cache from this analysis.	w.Designs.[Design1 Analysis].ClearModelCache()
GetDataName()	Get the name of the dataset. If an equation, this is parsed.	result = w.Designs.[Design1 Analysis].GetDataName()
RunAnalysis()	Run this analysis.	w.Designs.[Design1 Analysis].RunAnalysis()
SetDataName(bstr Name)	Set the dataset name the analysis will use.	w.Designs.[Design1 Analysis].SetDataName("Dataset Name")

Application

Syntax	Description	Example
Application()	Returns the current version of SystemVue.	Application()
Create(bstr Type, bstr Name)	Create a new object with the specified type and name.	result = Create("Notes", "ThisNote")
FileNewFromTemplate(bstr FileName)	Open a template from the template directory.	FileNewFromTemplate("Data Flow Template.wsv")
FileOpen(bstr FileName)	Open a file from the last opened directory.	FileOpen("Data Flow Template.wsv")
FileOpenExample(bstr FileName)	Open an example from the last opened directory.	FileOpenExample("SPDT.wsv")
FileOpenRecent(int FileNumber)	Open a recent file. 1 represents the most recent file.	Application.Manager.FileOpenRecent(1)
GetToolbarSet()	Gets the toolbar set as an object.	result = Application.Manager.GetToolbarSet()
GetWorkspaceByIndex(int iNum)	Returns the workspace object at index iNum	w=theApp.GetWorkspaceByIndex(0)
GetWorkspaceCount()	Returns the number of workspaces opened the instance of SystemVue	result =Application.Manager.GetWorkspaceCount()
OpenWorkspace(bstr strFile)	Loads the specified workspace without a open window prompt	OpenWorkspace("SPDT.wsv")
PostCommand(bstr CmdMsg)	Display a command message in the current view.	Application.Manager.PostCommand("fit_windows")
SaveTextToFile(bstr FileName, bstr ToShow)	Save text to a file.	SaveTextToFile "File.txt", "HelloWorld"
SetNetworkReuse(int iPorts)	Displays dialog to re use a design as a part with the specified number of ports. A part of iPort number of ports is created representing the design you select.	Application.Manager.SetNetworkReuse 2
Show(bstr ToShow)	Display the text in the Status box below the Edit box of a window.	Show("HelloWorld")
ShowToolBar(bstr ToolBarName, int Show)	Toggle, show, or hide a toolbar by name. 0 = Off, 1 = On	Application.Manager.ShowToolBar "Schematic", 1
Update()	Runs all pending analyses.	Application.Manager.Update()
ViewDesignSelector(int Flags)	Toggle (0), show (1), or hide (2) the Library Selector.	Application.Manager.ViewDesignSelector(0)
ViewPartPicker(int Flags)	Toggle (0), show (1), or hide (2) Part Selector A.	Application.Manager.ViewPartPicker(0)
ViewPartPickerB(int Flags)	Toggle (0), show (1), or hide (2) Part Selector B.	Application.Manager.ViewPartPickerB(0)
ViewSimulationStatus(int Flags)	Toggle (0), show (1), or hide (2) the Simulation Status window.	Application.Manager.ViewSimulationStatus(0)
ViewTuneWindow(int Flags)	Toggle (0), show (1), or hide (2) the Tune window.	Application.Manager.ViewTuneWindow(0)
ViewWorkspaceWindow(int Flags)	Toggle (0), show (1), or hide (2) the Workspace window.	Application.Manager.ViewWorkspaceWindow(0)

Atom

Syntax	Description	Example
ExportXML(bstr Path)	Save an object's XML stream to file. Path needs file extension.	w.Note.ExportXML "test.xml"
GetName()	Get the external name of an object.	result = w.GetName
ToString()	Convert an object into a string (text) representation.	result =w.Note.ToString
ToXML()	Convert an object into XML.	result =w.Note.ToXML
SetName(bstr bsName)	Set the object name	w.Note.SetName "HelloName"

Dataset

Syntax	Description	Example
ExportS(bstr FileName)	Export data as an S-parameter data file.	w.Designs.Design1_Data.ExportS("Design1_Data.s2p")
SnapShotToData	Creates a snapshot of a dataset or equation. This can be used to make a checkpoint dataset.	w.Designs.[Design1_Data].Eqns.SnapShotToData("New_Dataset")
DeleteAnalysisVars	Delete all calculated-by-analysis variables from a dataset	w.Designs.[Design1_Data].DeleteAnalysisVars()

Folder

Syntax	Description	Example
DeleteObject(bstr ObjectName)	Delete a SystemVue object.	w.Designs.DeleteObject("Design1_Data")
GetNameList(variant* ItemList)	Get the list of names in the folder. List is a collection of names.	w.Designs.GetNameList result
GetObjectCount()	Count the number of sub objects in the folder.	result = w.Designs.GetObjectCount()
GetObjectList(variant* ItemList)	Get the list of objects (as pointer).	w.GetObjectList result
GetObjectType(bstr ObjectName)	Gets the type of an object called ObjectName inside a folder.	w.Designs.GetObjectType("Signal")

Item

Syntax	Description	Example
AddProperty(IDispatch* Property)	Insert this property. Input must be an item.	result = w.GetItemByName("Note") w.Designs.AddProperty(result) 'adds the note to the designs folder
DeleteProperty(bstr Property)	Delete this property.	w.DeleteProperty("Note")
GetItemByIndex(int Index)	Get items by index starting with index 0.	result = w.GetItemByIndex(1)
GetItemByName(bstr ItemName)	Get item by name.	result = w.Designs.GetItemByName("Signal")
GetItemCount()	Count the number of items.	result = w.GetItemCount()
GetMethodList()	Get the list of methods from the GDISP entries.	result = w.GetMethodList()
GetParentOfItem(IDispatch* Child)	Get the parent item of given item.	child = w.Designs.[Design1 Analysis].DataName result = w.GetParentOfItem(child)
GetPropertyList(variant* ItemList)	Get the property list of an item.	w.Designs.Design1.PartList.GetPropertyList result
GetPropertyType(bstr PropName)	Get property type by name.	result=w.Designs.GetPropertyType("Design1 Analysis")
GetType()	Gets the type of an item.	result = w.Designs.Spectrum.GetType()
GetPropertyAsArray(bstr name, variant* ItemList)	Gets the contents of a property as a list	w.GetPropertyAsArray "Notes", result
GetVarCount()	Count the number of variables.	result = w.Designs.Spectrum.GetVarCount()
GetVarName(int Index)	Get the variable name at given index.	result = w.Designs.Spectrum.GetVarName(2)
GetVarType(int Index)	Get variable type at given index.	result = w.Designs.Spectrum.GetVarType(2)
GetVarValue(int Index)	Get the variable value at a given index.	result = w.Designs.Spectrum.Width.GetVarValue(2)
GetVarXMLName(int Index)	Get the XML name of a variable at a given index.	result = w.Designs.Spectrum.Width.GetVarXMLName(2)

HasProperty(bstr PropName)	Returns -1 is the Item has the property and 0 if it does not	if w.HasProperty("IsOpen") then
		Show "yes" 'is the workspace open?
		end if

SetProperty(bstr Property, variant* Value)	Set a property to a value.	number ="2500" w.Designs.Design1.PartList.S1.ParamSet.Frequency.SetProperty "DataEntry", number

Library

Syntax	Description	Example
GetPartList(variant* ItemList)	Get the part list of a library.	dim Symbols Library = w.GetLibrary("Design", "SymbolsQtr") Library.GetPartList Symbols For each part in Symbols Show part next

Menu

Syntax	Description	Example
Execute()	Execute a menu entry.	Application.Menu.File.New.Execute()
InsertItem(int Pos, bstr Text, bstr Name, bstr Script)	Insert a menu item inside any menu. The action of this new menu item is based on the script passed in.	Application.Menu.Run.InsertItem 0, "Open", "Open", "Application.Manager.OpenWorkspace(""SPDT.wsv"")"
InsertMenu(int iPosition,bstr bsText,bstr bsName)	Insert a menu tab on the top of the window.	Application.Menu.InsertMenu 8, "Run Script", "Run Script"
InsertSeparator(int Pos)	Insert a separator (bar). 0 is the initial position.	Application.Menu.Tools.InsertSeparator(2)

Parameters

Syntax	Description	Example
Get()	Get the parameter entry (what the user typed).	result = w.Designs.Design1.PartList.S1.ParamSet.Frequency.Get()
GetData	Get the formatted value of data.	result = w.Designs.Design1.PartList.S1.ParamSet.Frequency.GetData()
GetValue	Get the value of the data. This will always be in MKS for united Parameters.	result = w.Designs.Design1.PartList.S1.ParamSet.Frequency.GetValue()
Set(bstr NewValue)	Set the parameter entry (as if you typed it).	w.Designs.Design1.PartList.S1.ParamSet.Frequency.Set(7e3)
SetValue(variable)	Set the value of the data to the variable value.	a=7000 w.Designs.Design1.PartList.S1.ParamSet.Frequency.SetValue(a)

Part

Syntax	Description	Example
ChangeModel(bstr strName)	Change the Model of a part.	w.Designs.Design1.PartList.S1.ChangeModel("RampGen@Data Flow Models")
ChangeSymbol(bstr strName)	Change the Symbol of a part.	w.Designs.Design1.PartList.S1.ChangeSymbol("SYM_RampGen")
SetCustomValue(bstr ParamName, variant* piParamValue, bstr bsUnit, bstr bsValidate, bool vbShow)	Adds a custom value to a part, which will appear in the custom tab of the part properties.	dim val Freq2 = 1000 w.Designs.Design1.PartList.S1.SetCustomValue "Frequency 2", Freq2, "Hz", "Error", 1
Set(bstr bsParamValue)	Sets a part parameter to the specificed value.	' Set Sch1.C1.Output to 3.14 WsDoc.Designs.Sch1.PartList.C1.ParamSet.Output.Set("3.14")

Schematic

Syntax	Description	Example
AddAnnotationBox(bstr bsTag, bstr bsText)	Adds a text box named bsTag containing the text bsText to a schematic.	w.Design.BRIDGE_T.AddAnnotationBox "Hello", "World"
ExportIFF(bstr FileName)	Export to ADS/IFF format.	w.Design.BRIDGE_T.ExportIFF "IFF.iff"
GetIntent()	Get intended-use of the design. Returns integer value: 0=General, 1=Schematic, 2=Layout, 3=Symbol, 4=Model, 5=Footprint.	result = w.Design.BRIDGE_T.GetIntent()
PlacePart(bstr bstr int int int int)	Returns an HRESULT (0 indicates success).	Dim MySch MySch = w.Designs.Sch1.Schematic ' destination schematic MySch.PlacePart "Const@MyLibrary", "C1", 1500, 500, 0, 0
PlaceWire (bstr bsNetName, int X1, int Y1, int X2, int Y2)	Places a connecting wire starting at X1,Y1 and ending at X2, Y2. Use a NetName of "" for an unnamed (numeric) net. Returns an HRESULT.	w.Designs.Sch1.Schematic.PlaceWire "Net7", 2500, 2250, 2500, 3000
SelectObject(int X, int Y)	Selects the object at a specific point on the schematic. Coordinates are in 1000th of an inch. Returns an HRESULT.	w.Designs.Sch1.Schematic.SelectObject 2500, 3000
SetNet(bstr bNetName)	Sets the Net Name of a selected connecting wire. Returns an HRESULT.	result = w.Design.Sch1.Schematic.SetNet("Net9")
DeleteSelection()	Deletes currently selected objects from the schematic. Returns an HRESULT.	result = w.Design.Sch1.Schematic.DeleteSelection()

Equations

Syntax	Description	Example
SnapShotToData	Convert the equation variable values into a fixed dataset.	w.Equations.SnapShotToData
Calculate	Calculate an equation set. Designed for non-auto-calc equations. This is useful for running communications, for example.	result = w.Equations.Calculate()

Script

Syntax	Description	Example
RunScript(int Language)	Executes a script in the specified language. 0==VBScript, 1==JScript	w.Script1.RunScript(0)

Workspace

Syntax	Description	Example
ClearCompareLog()	Clears the Run and Compare error log.	w.ClearCompareLog()
ClearErrorLog()	Clears the error log at the bottom of the window.	w.ClearErrorLog()
GetCompareLog()	Gets the compare as generated by the RunAndCompare function.	SaveTextToFile "test.txt",w.GetCompareLog
DeleteOldDatasets()	Deletes all but the most recent dataset. Works if you have used the RunAndCompare function.	w.DeleteOldDatasets
IsAnalysisDone()	Returns 1 if the analysis is done and 0 if it is not.	result = w.IsAnalysisDone()
RunAndCompare(int numErrors, double dTolerance, double dAbsouleTol)	Runs and creates a new dataset for each analysis. Compares the two datasets on a tolerance of dTolerance and ignores any values below the absolute tolerance dAbsoluteTol.It reports errors stopping after numErrors errors have been found to the compare log. The return value is Pass, Warn, or Fail	w.RunAndCompare 2, 0.02, 0.00001
SaveErrorLog(bstr FileName)	Save the error log into a file.	w.SaveErrorLog("ErrorLog.txt")
Save()	Save a workspace.	w.Save()
SaveAs(bstr FileName)	Save a workspace with new name.	w.SaveAs("name.wsv")

Using S-Parameters in SystemVue (RF Design Kit)

This section shows how S-Parameter data can be incorporated into SystemVue designs and exported to other programs.

S-Parameters are commonly used in RF circuits to represent incident and reflected traveling waves. S-Parameters are created by a linear simulation and are generally available from component manufactures. Several libraries of S-Parameter data ship with SystemVue. S-Parameters in SystemVue are imported and exported in the Touchstone format.

Contents

Creating S-Parameter Data

1. Create the schematic for which the S-Parameter data will be represented
2. Add a linear analysis and point it to the desired schematic
3. Set the frequency range and step size of the linear analysis to the desired resolution of the S-Parameter data
4. Run the linear analysis
5. Export linear analysis data as a S-Parameter file

Using S-Parameters in a Simulation

The use model for S-Parameters manually imported into the workspace versus file based S-Parameter is slightly different. The model used in the schematic determines how the S-Parameters will be managed.

Displaying S-Parameter Data

The easiest way to display S-Parameter data is to open up the S-Parameter dataset and the right click on the **S** variable. Then select **Create Table** or **Graph** and the type of graph. The data will automatically be displayed.

File Based S-Parameters

File based S-Parameter import the S-Parameters from a file into a dataset providing simulation cache. This dataset is used when reloading the workspace to re-cached the data. If the dataset is deleted the S-Parameters will be re-imported the next time a simulation needs the data.

1. Place a S-Parameter file based part in the schematic (1-port , 2-port , n-port). This can be done from the Linear Toolbar or the Part Selector
2. Double click the part to bring up the part properties
3. Click the **Browse** button to browse to the S-Parameter file
4. Add an analysis and point it to the desired schematic
5. Run the analysis

Physical S-Parameters

S Parameters can be taken or formed in such a way that they represent non physical parts like negative resistors. Realistic real world answers only come when S-Parameters are physical. If S-parameters are physical, then the corresponding Y-parameters will meet **all** of the following requirements:

1. The real part of every diagonal entry must be positive. i.e. **Real.Yp[i,i] > 0**
2. The real part of every non-diagonal entry must be negative. i.e. **Real.Yp[i,j] < 0** where i is not equal to j
3. The absolute value of the row real summation, excluding the diagonal, must be less than real value of the diagonal in that row. i.e. **abs (sum(Real.Yp[i,j])) < Real.Yp[i,i] where i is not equal to j**
4. The absolute value of the column real summation, excluding the diagonal, must be less than the real value of the diagonal in that column. i.e. **abs (sum(Real.Yp[i,j])) < Real.Yp[j,j] where i is not equal to j**

Note: It is assumed the Y parameters are in Real _ j Imaginary format.

Examples:
Here are some typical Y-parameters (which is converted from S-parameters):

The Y parameter matrix for F = 3000 is:

0.077 - j0.122   -0.078 + j0.123

-0.078 + j0.123   0.078 - j0.121

This matrix meets items 1 and 2 but not 3 and 4, because abs(Real.Y[1,2]) > Real.Y[1,1] or abs(Real.Y[2,1]) > Real.Y[1,1], so these S parameters are **non physical**.

Touchstone Format

These files contain small-signal S-parameters described by frequency-dependent linear network parameters for 1- to 10-port components. The 2-port component files can also contain frequency-dependent noise parameters. This data file format is also known as Touchstone format.

Overview

Touchstone files are ASCII text files in which frequency dependent data appears line by line, one line per data point, in increasing order of frequency. Each frequency line consists of a frequency value and one or more pairs of values for the magnitude and phase of each S-parameter at that frequency. Values are separated by one or more spaces, tabs or commands. Comments are preceded by an exclamation mark (!). Comments can appear on separate lines, or after the data on any line or lines. Extra spaces are ignored.

Filename Recommendations

1-port: filename.s1p, 2-port: filename.s2p, ... i.e. n-port: filename.snp

Basic File Format

The file format consists of:

- Comments
- Option Line
- S-Parameter Data Lines
- Noise Data Lines

Comments

Comments can be placed anywhere in the file by preceding a comment with the exclamation mark !. A comment can be the only entry on a line or can follow the data.

The Option Line

The option line specifies the format of the data in the file. The line looks like: # GHZ S MA R 50

```
#  <FREQ_UNITS>  <TYPE>  <FORMAT>  <Rn>
#  = Option line delimiter
```

```
<FREQ_UNITS> = Units of the frequency data. Options are GHz, MHz, KHz, or Hz.
<TYPE> = Type of file data. Options are:  S, Y or Z for S1P components, S, Y, Z, G, or H for S2P
components, S for 3 or more ports
<FORMAT> = S-parameter format. Options are: DB for dB-angle, MA for magnitude angle, RI for real-
imaginary
<Rn> = Reference resistance in ohms, where n is a positive number. This is the impedance the S-
parameters were normalized to.
```

In summary:
For .s1p files:

```
#  [ HZ / KHZ / MHZ / GHZ]  [ S / Y / Z ]  [ MA / DB / RI ]  [ Rn ]
```

For .s2p files:

```
#  [ HZ / KHZ / MHZ / GHZ ]  [ S / Y / Z / G / H ]  [ MA / DB / RI ]  [ Rn ]
```

For .snp (n >= 3) files:

```
#  [ HZ / KHZ / MHZ / GHZ ]  [ S ]  [ MA / DB / RI ]  [ Rn ]
```

where square brackets [...] indicate optional information; .../.../.../ indicates that you select one of the choices; and, n is replaced by a positive number.

S-Parameter Data

Frequency data lines contain the data of interest. A special format is used for 2-port data files where all of the network parameter data for a single frequency is listed on one line. The order of the network parameters is:

S11, S21, S12, S22

For 3-port or higher data files, the network parameters appear in the file in a matrix form, each row starting on a separate line. A maximum of four network parameters (with 2 real numbers for each) appear on any line. The remaining network parameters are continued on as many additional lines as are needed.

The following sections describe the data-line format for single and multi-port components.

S-Parameter Data Line Format

The frequency line data will have one of the following formats:

Magnitude Angle

```
<FREQ>  |S11|  <S11  |S21|  <S21  |S12|  <S12  |S22|  <S22
```

Real Imaginary

```
<FREQ>  Re{S11}  Im{S11}  Re{S21}  Im{S21}  Re{S12}  Im{S12}  Re{S22}  Im{S22}
```

dB Angle

```
<FREQ>  20log10|S11|  <S11  20log10|S21|  <S21  20log10|S12|  <x12 20log10|S22|  <S22
```

Note: For each s1p and s2p file format, the data must be on one line.

3-port Data Magnitude Angle Example

```
<FREQ>  |S11|  <S11  |S12|  <S12  |S13|  <S13
       |S21|  <S21  |S22|  <S22  |S23|  <S23
       |S31|  <S31  |S32|  <S32  |S33|  <S33
```

4-port Data Magnitude Angle Example

```
<FREQ>  |S11|  <S11  |S12|  <S12  |S13|  <S13  |S14|  <S14
       |S21|  <S21  |S22|  <S22  |S23|  <S23  |S24|  <S24
       |S31|  <S31  |S32|  <S32  |S33|  <S33  |S34|  <S34
       |S41|  <S41  |S42|  <S42  |S43|  <S43  |S44|  <S44
```

Noise Parameters

Noise parameters can be included in Touchstone files. Noise data follows the S-parameter data. It has the following format:

<FREQ> <NF_{min} dB> <|$Gamma_{opt}$|> <Ang($Gamma_{opt}$)> <Rn>

where

<FREQ> = Frequency of the noise data In units specified in the options line
<NF_{min} dB> = Minimum noise figure in dB
<|$Gamma_{opt}$|> = Magnitude of the source reflection coefficient at the minimum noise figure
<Ang($Gamma_{opt}$)> = Phase angle in degrees of the source reflection coefficient at the minimum noise figure

<Rn> = Effective noise resistance normalized to the system impedance defined in the option line. It defines the rate at which the noise figure increases as the reflection coefficient is moved away from the optimum values. In other words, how tightly spaced the noise circles are.

Note: The frequencies for noise and S parameters need not match. The only requirement is that the lowest noise-parameter frequency be less than or equal to the highest S-parameter frequency.

Noise Data Example

This is an example of a data file with noise data:

```
! NEC710
# GHZ S MA R 50
2    .95   -26  3.57  157  .04  76  .66  -14
22  .60  -144  1.30   40  .14  40  .56  -85
! NOISE PARAMETERS
4    .7    .64   69  .38
18  2.7   .46  -33  .40
```

S-Parameter 5 to 99 Port File Formats

These file formats appear in a matrix form similar to the 3 and 4 port files, except that only four S-parameters (with 2 real numbers for each) can appear on a given line. Therefore, the remaining S-parameters in that row of the S-matrix continue on the next line of the file.

Each row of the S-matrix must begin on a new line of the file. The first line of the first row of the S-matrix begins with the frequency value.

S-Parameter 10-Port File Example (at One Frequency)

```
#  <FREQ_UNITS>  <TYPE>  <FORMAT>  <Rn>
<FREQ> magS11 angS11 magS12 angS12 magS13 angS13 magS14 angS14    ! 1st row
magS15 angS15 magS16 angS16 magS17 angS17 magS18 angS18
magS19 angS19 magS1,10 angS1,10
```

```
magS21 angS21 magS22 angS22 magS23 angS23 magS24 angS24    ! 2nd row
magS25 angS25 magS26 angS26 magS27 angS27 magS28 angS28
magS29 angS29 magS2,10 angS2,10
magS31 angS31 magS32 angS32 magS33 angS33 magS34 angS34    ! 3rd row
magS35 angS35 magS36 angS36 magS37 angS37 magS38 angS38
magS39 angS39 magS3,10 angS3,10
magS41 angS41 magS42 angS42 magS43 angS43 magS44 angS44    ! 4th row
magS45 angS45 magS46 angS46 magS47 angS47 magS48 angS48
magS49 angS49 magS4,10 angS4,10
magS51 angS51 magS52 angS52 magS53 angS53 magS54 angS54    ! 5th row
magS55 angS55 magS56 angS56 magS57 angS57 magS58 angS58
magS59 angS59 magS5,10 angS5,10
magS61 angS61 magS62 angS62 magS63 angS63 magS64 angS64    ! 6th row
magS65 angS65 magS66 angS66 magS67 angS67 magS68 angS68
magS69 angS69 magS6,10 angS6,10
magS71 angS71 magS72 angS72 magS73 angS73 magS74 angS74    ! 7th row
magS75 angS75 magS76 angS76 magS77 angS77 magS78 angS78
magS79 angS79 magS7,10 angS7,10
magS81 angS81 magS82 angS82 magS83 angS83 magS84 angS84    ! 8th row
magS85 angS85 magS86 angS86 magS87 angS87 magS88 angS88
magS89 angS89 magS8,10 angS8,10
magS91 angS91 magS92 angS92 magS93 angS93 magS94 angS94    ! 9th row
magS95 angS95 magS96 angS96 magS97 angS97 magS98 angS98
magS99 angS99 magS9,10 angS9,10
magS10,1 angS10,1 magS10,2 angS10,2 magS10,3 angS10,3 magS10,4 angS10,4     ! 10th row
magS10,5 angS10,5 magS10,6 angS10,6 magS10,7 angS10,7 magS10,8 angS10,8
magS10,9 angS10,9 magS10,10 angS10,10
```

Linear 1-Port (.s1p) File Example

```
# GHZ   S   RI   R   50.0
    1.00000000    0.9488    -0.2017
    1.50000000    0.9077    -0.3125
    2.00000000    0.8539    -0.4165
    2.50000000    0.7884    -0.5120
    3.00000000    0.7124    -0.5978
    3.50000000    0.6321    -0.6546
    4.00000000    0.5479    -0.7013
    4.50000000    0.4701    -0.7380
    5.00000000    0.3904    -0.7663
    5.50000000    0.3302    -0.7778
    6.00000000    0.2702    -0.7848
    6.50000000    0.2041    -0.7890
    7.00000000    0.1389    -0.7878
    7.50000000    0.0894    -0.7849
    8.00000000    0.0408    -0.7789
    8.50000000    0.0134    -0.7649
    9.50000000    0.0654    -0.7471
    9.00000000    0.1094    -0.7319
    10.0000000    0.1518    -0.7140
```

Linear 2-Port (.s2p) File Example

```
# GHZ    S   RI   R   50.0
1.0000  0.3926  -0.1211  -0.0003  -0.0021  -0.0003  -0.0021  0.3926  -0.1211
2.0000  0.3517  -0.3054  -0.0096  -0.0298  -0.0096  -0.0298  0.3517  -0.3054
10.000  0.3419   0.3336  -0.0134   0.0379  -0.0134   0.0379  0.3419   0.3336
! Noise parameters
 1.0000  2.0000  -0.1211  -0.0003  .4
 2.0000  2.5000  -0.3054  -0.0096  .45
 3.0000  3.0000  -0.6916  -0.6933  .5
 4.0000  3.5000  -0.3756   0.4617  .55
 5.0000  4.0000   0.3880   0.6848  .6
 6.0000  4.5000   0.0343   0.0383  .65
 7.0000  5.0000   0.6916   0.6933  .7
 8.0000  5.5000   0.5659   0.1000  .75
 9.0000  6.0000   0.4145   0.0307  .8
10.0000  6.5000   0.3336   0.0134  .85
```

Linear 3-Port (.s3p) File Example

```
#  GHZ  S  MA  R  50.0
!  POWER DIVIDER, 3-PORT
5.00000  0.24254  136.711   0.68599  -43.3139  0.68599  -43.3139     ! Frequency Line 1
         0.68599  -43.3139  0.08081   66.1846  0.28009  -59.1165
         0.68599  -43.3139  0.28009  -59.1165  0.08081   66.1846
6.00000  0.20347  127.652   0.69232  -52.3816  0.69232  -52.3816     ! Frequency Line 2
         0.69232  -52.3816  0.05057   52.0604  0.22159  -65.1817
         0.69232  -52.3816  0.22159  -65.1817  0.05057   52.0604
7.00000  0.15848  118.436   0.69817  -61.6117  0.69817  -61.6117     ! Frequency Line 3
         0.69817  -61.6117  0.02804   38.6500  0.16581  -71.2358
         0.69817  -61.6117  0.16581  -71.2358  0.02804   38.6500
```

Linear 4-Port (.s4p) File Example

```
#  GHZ  S  MA  R  50
5.00000 0.60262  161.240  0.40611 -42.2029 0.42918 -66.5876 0.53640 -79.3473     ! Frequency Line
1
        0.40611  -42.2029 0.60262 161.240  0.53640 -79.3473 0.42918 -66.5876
        0.42918  -66.5876 0.53640 -79.3473 0.60262 161.240  0.40611 -42.2029
        0.53640  -79.3473 0.42918 -66.5876 0.40611 -42.2029 0.60262 161.240
6.00000 0.57701  150.379  0.40942 -44.3428 0.41011 -81.2449 0.57554 -95.7731     ! Frequency Line
2
        0.40942  -44.3428 0.57701 150.379  0.57554 -95.7731 0.41011 -81.2449
        0.41011  -81.2449 0.57554 -95.7731 0.57701 150.379  0.40942 -44.3428
        0.57554  -95.7731 0.41011 -81.2449 0.40942 -44.3428 0.57701 150.379
7.00000 0.50641  136.693  0.45378 -46.4151 0.37845 -99.0918 0.62802 -114.196     ! Frequency Line
3
        0.45378  -46.4151 0.50641 136.693  0.62802 -114.196 0.37845 -99.0918
        0.37845  -99.0918 0.62802 -114.196 0.50641 136.693  0.45378 -46.4151
        0.62802 -114.196  0.37845 -99.0918 0.45378 -46.4151 0.50641 136.693
```

See also *1-Port* (rfdesign), *2-Port* (rfdesign), *3-Port* (rfdesign), *4-Port* (rfdesign), and *n-Port* (rfdesign) S-parameter models

Sweeps

Sweeps are used to create analysis results that are functions of a parameter tuned at several values. A sweep is an evaluation object that controls a specific analysis. It also contains a single tuned parameter that is swept across a range of values specified by the user.

Once a parameter is made tunable (part parameter 'Tune' checkbox has been checked or a question mark '?' has been placed in front of the equation value) it can be selected in the **Parameter Sweep Properties** dialog box. The user specifies hard **Start** and **Stop** values as well as the number of swept points.

The number of swept points can be specified in 1 of 4 ways:

- **Linear: Number of Points**
- **Log: Points / Decade**
- **Linear: Step Size**
- **List**

Once the **analysis** has been selected and the **swept parameter** has been defined the **sweep** will tune the parameter to the first point, run the analysis, and then save the data in a sweep dataset. The swept parameter is tuned to the next value, the analysis is re-ran and the new data will be appended to prior values. This process repeats until the last swept point is reached.

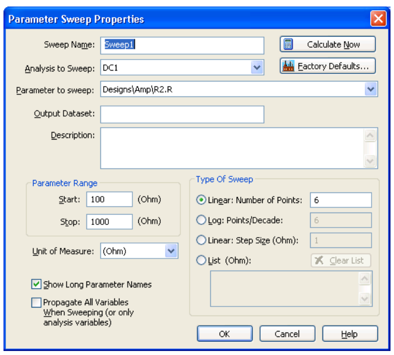
Hint
Remember, an **analysis** and **tuned variable** must exist in the workspace before a sweep can be created.

Note
A sweep can control another sweep. Sweeps are also considered an **analysis** and will appear in the **Analysis to Sweep** list.

Note
For additional information on sweeps in Spectrasys see *Sweeps of a Path* (sim).

Contents

Parameter Sweep Properties

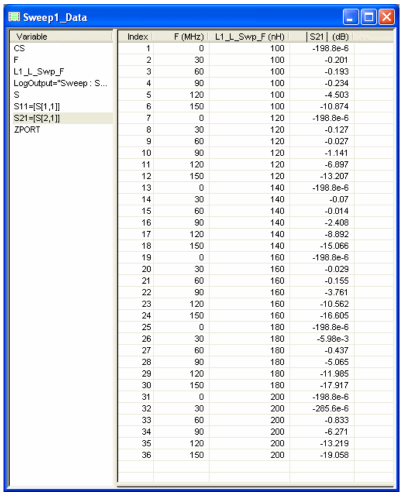


Name	Description
Sweep Name	Name of Sweep Evaluation
Analysis to Sweep	Analysis used for the parameter sweep. The selected analysis will be recalculated for each different value of the swept parameter.
Parameter to Sweep	Parameter that gets changed to create the sweep. All parameters defined as tunable are available to be swept.
Output Dataset	Dataset file in which the data is saved. If not specified, the dataset name will be the name of the analysis with "_Data" appended.
Description	Description of the evaluation being run. For documentation purposes only, not otherwise used by SystemVue.
Calculate Now	Run the evaluation. Always runs the analysis, regardless of whether or not any changes were made.
Propagate All Variables When Sweeping	User created variables in the source dataset will be swept and aggregated into the sweep dataset.
Show Long Parameter Name	Display the full parameter name (with path) in case you have multiple parameters with the same short name (such as C1.C).
Factory Defaults	Reset all values to their default
Parameter Range	Start. The lower bound (minimum frequency) of the sweep.
Stop	The upper bound (maximum frequency) of the sweep.
Unit of Measure	Unit of measure used for the evaluation
Type of Sweep	Linear: Number of Points. Number of points in entire sweep.
Log: Points/Decade	Number of points in each decade of the sweep.
Linear: Step Size (MHz)	Allows specification of start and stop frequencies, and space between points.
List of Frequencies (MHz)	Allows the explicit specification of analysis frequencies. These points are entered into the List of Frequencies box separated by spaces.

Understanding Swept Data

The following figure shows the dataset for a modified version of the *Getting Started with Parameters Sweeps* (users) example. This example was modified to reduce the number of linear simulation points from 101 to 6 so the results could be seen in a single figure.

In summary, there is a linear simulation of simple LC low pass filter over 6 frequency points (0, 30, 60, 90, 120, and 150 MHz). A sweep has been created that tunes the filter inductor parameter **L** across 6 values (100, 120, 140, 160, 180, and 200 nH).



Index	F (MHz)	L1_L_Swp_F (nH)	|S21| (dB)
1	0	100	-198.8e-6
2	30	100	-0.201
3	60	100	-0.193
4	90	100	-0.234
5	120	100	-4.503
6	150	100	-10.874
7	0	120	-198.8e-6
8	30	120	-0.127
9	60	120	-0.027
10	90	120	-1.141
11	120	120	-6.897
12	150	120	-13.207
13	0	140	-198.8e-6
14	30	140	-0.07
15	60	140	-0.014
16	90	140	-2.408
17	120	140	-8.892
18	150	140	-15.066
19	0	160	-198.8e-6
20	30	160	-0.029
21	60	160	-0.155
22	90	160	-3.761
23	120	160	-10.562
24	150	160	-16.605
25	0	180	-198.8e-6
26	30	180	-5.98e-3
27	60	180	-0.437
28	90	180	-5.065
29	120	180	-11.985
30	150	180	-17.917
31	0	200	-198.8e-6
32	30	200	-285.6e-6
33	60	200	-0.833
34	90	200	-6.271
35	120	200	-13.219
36	150	200	-19.058

Column Name	Description
Index	Row number in the table.
F (MHz)	First independent variable of swept data. In this example, this is the frequency range of the linear analysis. It is repeated for each tuned value of the swept parameter.
L1_L_Swp_F (nH)	Second independent variable of swept data. In this example, this is the tuned value of the inductor parameter **L** at each frequency point of the linear analysis.
|S21| (dB)	Dependent variable for each tuned parameter value (inductor) at each frequency.

Note
The independent variables (in this example **F** and **L1_L_Swp_F**) are also stored in the dataset.

Hint
For additional information on indexing into sweeps see *Using Math Language* (users).

Getting Started with Parameter Sweeps

To add a Parameter Sweep Evaluation:

1. Create a *design* (users) with a schematic.
2. Define your tunable parameters.
3. Click the New Item button ( ) on the Workspace Tree toolbar and choose "Add Sweep" from the Evaluation menu.
4. Define the *Parameter Sweep Properties* (users) and click **OK**. The analysis runs and creates a data set.

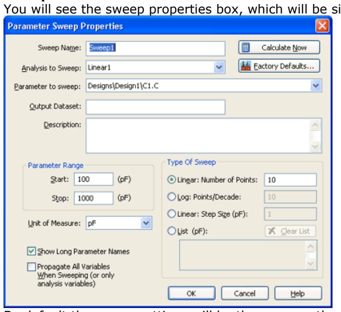
For advanced applications, you can nest Parameter sweeps, creating 4-D, 5-D, or higher data. This data can then be viewed on a table.

Performing a Parameter Sweep

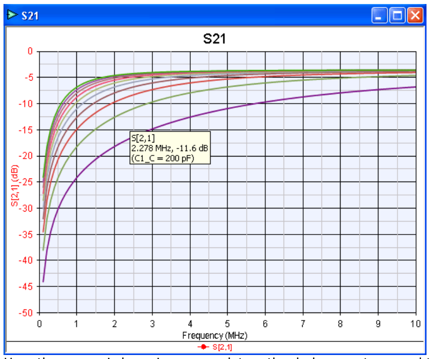
A parameter sweep gives you a set of responses for a set of parameter values. You can perform a parameter sweep on any tuned variable.

To create a parameter sweep:

1. Click the New Item button ( ) on the Workspace Tree toolbar and select **Add Sweep** from the Evaluations menu.
2. You will see the sweep properties box, which will be similar to this:



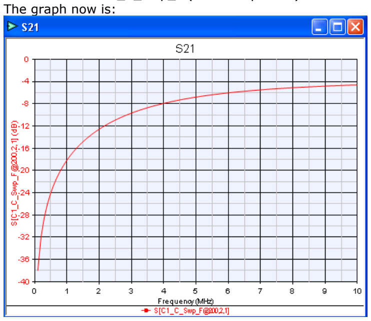
3. By default the sweep settings will be the same as the last time you created a sweep. The default parameter to sweep is just the first in the list. Here the parameter is in the Designs folder, in the design named Design1, in the part named C1, as parameter C.
   In the list are all tuned parameters (or equation variables). Use the settings shown above, then click Calculate Now to calculate the sweep.
4. Note that a Sweep1_Data dataset is built.
5. Double-click the S21 graph and change the "Default Dataset or Equations" to Sweep1_Data - so you plot S21 for the swept data. Turn off symbols by clicking the Symbols button (the last button on the Graph toolbar). You get a range of traces that looks like this:



Here the mouse is hovering over a dot on the dark green trace and the popup identifies the trace and value.

To look at the range at 200 pF enter the following formula into the graph line

6. Enter S[C1_C_Swp_F@200,2,1]. Note that the swept C value is in the Sweep1_Data set and named C1_C_Swp_F (C1.C swept on F).
7. The graph now is:

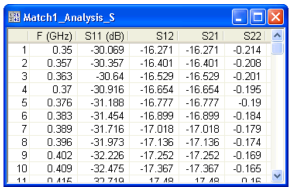


Tables

Tables provide text-based tabular output instead of graphical output. There is only one type of table in SystemVue. You can place any measurement in a table. Change the properties of a table using the Table Properties window.

Use Ctrl_MouseWheel to zoom in and out on tables.

Any type of alpha-numeric data (such as S-parameters) may be displayed in a table:



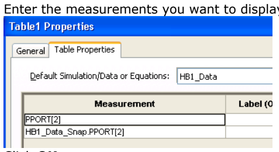
	F (GHz)	S11 (dB)	S12	S21	S22
1	0.35	-30.069	-16.271	-16.271	-0.214
2	0.357	-30.357	-16.401	-16.401	-0.208
3	0.363	-30.64	-16.529	-16.529	-0.201
4	0.37	-30.916	-16.654	-16.654	-0.195
5	0.376	-31.188	-16.777	-16.777	-0.19
6	0.383	-31.454	-16.899	-16.899	-0.184
7	0.389	-31.716	-17.018	-17.018	-0.179
8	0.396	-31.973	-17.136	-17.136	-0.174
9	0.402	-32.226	-17.252	-17.252	-0.169
10	0.409	-32.475	-17.367	-17.367	-0.165

Contents

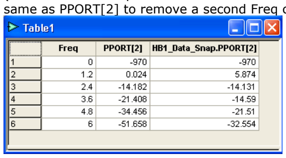
Creating Tables

- The easiest way to **create** a new graph is using the *Instagraph Feature* (users).
- The easiest way to **add** an arbitrary measurement to an existing table is via the *Graph Properties* (users).
- Tables can also be created manually:

1. Click the New Item button ( ) on the Workspace Tree toolbar and select **Add Table**.
2. Enter the measurements you want to display in the Table Properties dialog..



3. Click **OK**.
(**note** that the Independent Variable for PPORT2 in the Snap dataset was set to the same as PPORT[2] to remove a second Freq column)



	Freq	PPORT[2]	HB1_Data_Snap.PPORT[2]
1	0	-970	-970
2	1.2	0.024	5.874
3	2.4	-14.182	-14.131
4	3.6	-21.408	-14.59
5	4.8	-34.456	-21.51
6	6	-51.658	-32.554

The ***Label*** entry determines the column labels.

If the data is complex it will often display as dB or magnitude by default. To see the full complex data select a format from the ***Complex Format*** column.

If you want to print a table, copy it to a notes object by using the ***Copy To Notes*** right-click menu entry. This copies the headings and data into an HTML table which you can then copy to Word or other HTML editor or you can just print the Notes. Modify the Notes manually to change fonts or formatting.

New Right-click in the table header to get a popout menu that lets you turn off columns.

Templates

Templates are a very convenient way to get started quickly with a new design. They give you a complete circuit as your starting point. You can also modify templates for your specific program. Many templates are included with SystemVue, and you can easily add your own.

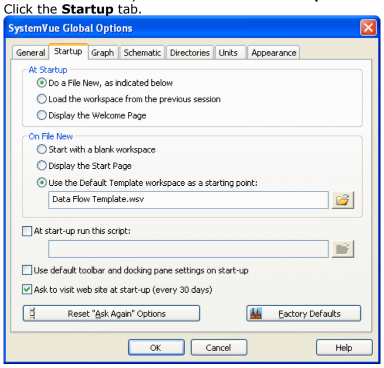
- *Selecting a SystemVue Template* (users)
- *Reviewing the SystemVue Templates* (users)

Selecting a SystemVue Template

The Default.wsx template is automatically loaded whenever a new workspace is created. You can use it or select a different SystemVue template.

To select a template to always start with:

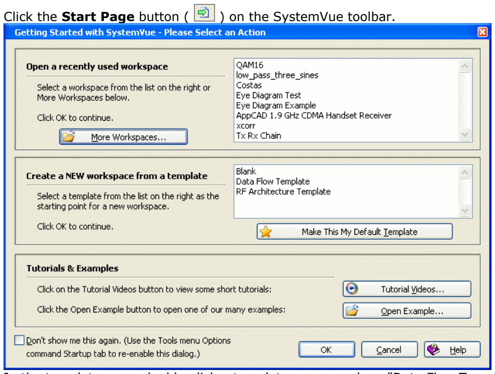
1. Click **Tools** on the SystemVue menu and select **Options**.
2. Click the **Startup** tab.



3. Click the **Start With This Template** button.
4. Click the folder button and select a template.
5. Click **Open**, and then click **OK**.

To select a template once:

1. Click the **Start Page** button ( ) on the SystemVue toolbar.



2. In the templates area double-click a template name, such as "Data Flow Template".

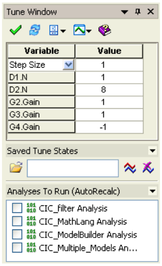
Reviewing the SystemVue Templates

The table below lists all of the workspace templates included in SystemVue.

File Name	Description
Blank.wsv	A blank schematic.
Data Flow Template.wsv	A data flow template.
RF Architecture Template.wsv	A template for working with RF architecture.

Tuning Variables

One of the most powerful features of SystemVue is real-time tuning of values in variables. You can use tuned variables almost anywhere in SystemVue, including part parameters. See almost any of our examples for tuned variables. Tuned variables are listed in the Tune Window as shown:



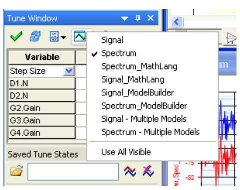
Any numeric parameter in a part can be made tunable. You can tune the value of a variable or use *Gang Tuning* (users) to adjust a value which is used in more than one place. SystemVue lets you dynamically tune variables to determine whether your design meets its requirements. You can do this by entering different values for a specific variable and simultaneously viewing the response in a graph. Continue adjusting values and viewing the graph until you get the desired response.

Contents

Checkpoints

A checkpoint is a saved intermediate point. In a graph, it is usually a dashed trace showing potentially good values.

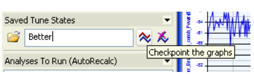
- Click the **Select Graphs** dropdown button and select (check) only the graphs you want checkpointed while tuning, as shown below



- Or, check **Use All Visible** to use all visible graphs. The graph list dynamically changes as you open and close graphs.

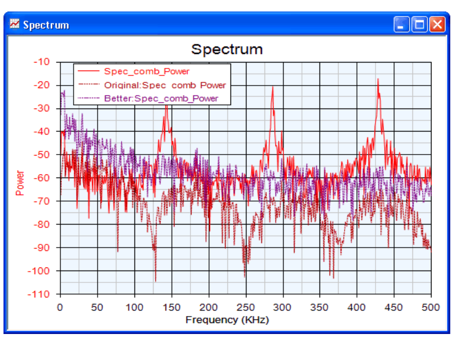
To establish checkpoints in a graphed analyses:

1. If the **Saved Tune States** panel is not currently visible, click its "unfold" button on the right.
2. Type a name into the Named Setting entry field (such as *Better*).



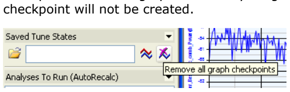
3. Click the Checkpoint button.

As you tune you will see an echo left behind of the original settings, this is the checkpoint. You can add as many checkpoints as you like. Each new checkpoint will have a dashed trace and be in a darker color.



To remove all checkpoint traces from all graphs:

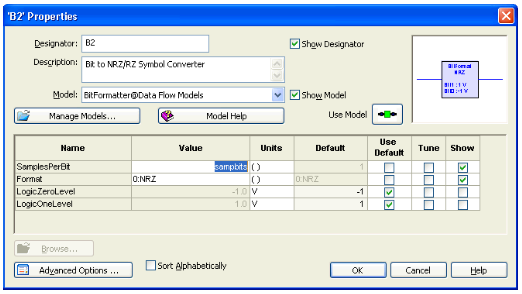
1. Click the Remove Checkpoint button in the tune window. This will remove checkpoints from the graphs listed in your graph checkpoint list. As you then tune a checkpoint will not be created.



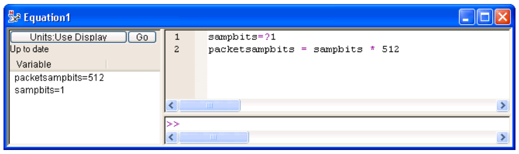
Gang Tuning

Another common task in tuning is to adjust more than one parameter at the same time. This is called Gang Tuning and the easiest way to do it is with an equation variable.

Start with an example: Signal Processing / CrossCorr. In the following figure, an equation variable has been setup for the "B2" BitFormatter. A new variable named *sampbits* has been entered, so that the both BitFormatters can **share the same value** for the SamplesPerBit parameter.



Then add an Equation to the workspace and then define the variable (and any others with might depend on it).

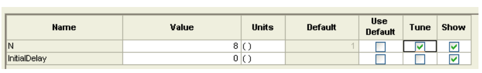


The variable *sampbits* is defined with a ?1. The 1 is the starting value and the ? syntax makes the variable tuneable. Other variables, such as *packetsampbits* can also be defined based on it (and other variables), for use elsewhere in the workspace.

For more information about Equations and setting variables tunable from Equations, please refer to the *Using Equations* (users) section.

Making a Part Parameter Tunable

1. Double-click a part on any schematic; this will bring up the Part Properties dialog.
2. Click the **Tune** check box next to any parameter you wish to be tunable.
3. Click **OK**



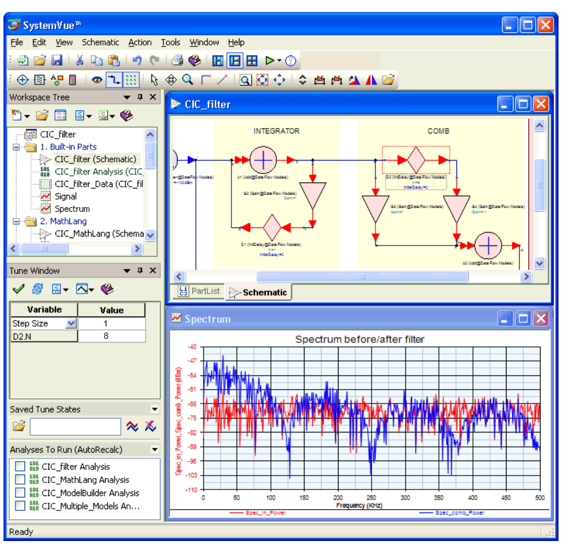
4. Notice that the variable(s) to be tuned have now changed color on the schematic and appear in the Tune window. To tune this parameter, use any of the methods discussed in *Setting Tuned Values* (users).

Note that part parameters can actually be selected for tuning in a number of ways:

- Double-click the part and check the Tune box next to a parameter value (as described above).
- Click the part to select it, then select **Make Components Tunable** from the Schematic menu. This sets the first parameter of the part to be tunable.
- Click the part to select it, then click the **Make Tunable** schematic toolbar button, which **toggles** the tunable setting of the first parameter.
- Parameters can be marked for tuning via on-screen editing: click a part parameter, unfold the parameters window (if necessary, use the "unfold" « button), and click in the first cell column. A 'T' indicates that a parameter is tunable.
- To mark many items tunable at once, try this approach:
  1. In the Tune Window, click the **Variable Options** button
  2. Select **Select Variables** from the drop down menu, which will show a comprehensive list of **everything** which can be tuned.
  3. Check the items you wish to tune.
  4. Click **OK**.

An example: Making D2.N tunable

1. First, load the **Model Building / CIC Filter** example and rearrange your screen so that it looks like the screenshot below.
2. Double-click part D2 and check the Tune box next to a parameter value.
3. Once D2 is tunable, the N=8 line should turn teal-colored and the Tune Window should now have a D2.N entry. All of the analyses will turn red because the schematic has changed.



Actually changing a value

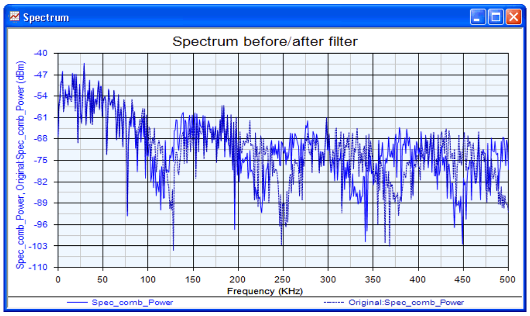
Tune the D2.N value by selecting it; click in the grid box where it says 8. Then, do any of the following:

- Roll the mouse-wheel to tune up or down
- Press the PgUp or PgDown key to **tune** up or down
- Click the up/down arrows in the grid box to **move** up or down

- Or type a new value and press enter

After typing a new value N and of 9 pressing enter, we see the following display:

- The dashed, dark blue trace is the original simulation result Spectrum. It is dashed because it is a "checkpointed" trace.
- The bright blue is the new, tuned simulation result.

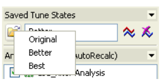


Save often

This is always a good idea. Saved workspaces remember all the tune settings, including what you were tuning, what runs when tuned, and which graphs to update. Click the save button (the diskette icon in the main toolbar) to save.

Reverting Tuned Values

- SystemVue lets you revert to your original values if you do not want to keep the newly tuned values.
- Click the **Use These Settings** button and select a named set. The set named Original is the original settings.



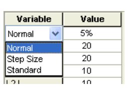
Tuning Options

To assist with the tuning of all the various types of variables you may need to adjust, there are a number of tuning options, which control how and what is tuned, when and what is updated, etc.

Specifying how values are tuned up/down

The tuning percentage controls the amount values are stepped when tuning. Whenever a variable is tuned up or down, a tuning ratio is used to calculate the new value. You can use the Tune window to adjust the Normal and Standard options by percentages. The Step Size option uses decimal values for adjusting.

1. Click the first box in the Variable column in the Tune window.
2. Select an option from the list. There are three options which control tuned variables values:
   - **Normal** – This option is steps the tuned value by the specified percentage, and is unrestricted. For lumped parts, such as resistors and capacitors, values between zero (0) and infinity are possible. You can use this option to determine the theoretical optimum values. This typically increments the value by 5%.
   - **Step Size** – This option adds or subtracts the specified step-size to the parameter. For example, if the step-size value equals 0.5, then the allowable parameter values are 0.5, 1.0, 1.5, and so on.
   - **Standard** – This option uses only standard values for lumped circuit elements, such as 1.2, 3.3, 4.7, 5.6, and so on. The tuning percentage is shown in the first box in the Value column. This controls the amount values are stepped when tuning.



Tip: You can press F6 to decrease or F7 to increase the tuning percentage by a factor of 2.

To checkpoint graphs, when you save a setting

- Click the Graph checkpoint button in the Tune Window and check on/off the graphs you want to checkpoint when you checkpoint a named setting. The menu goes away when you click off the menu area.
- Check **Use All Visible** to have the list be all visible graphs. The list dynamically changes as you open and close graphs.

Setting Tuned Values

You can set tuned values in a number of ways. Begin by selecting the value you want to change (click in the grid box cell holding the value), then

Click the scroll arrows:

- Click the up arrow to increase the value, the down arrow to decrease the value.
- The analyses you've selected will run will run automatically.

Scroll the mouse wheel:

- Roll the mouse wheel to scroll up and down to tune the value up and down.
- The analyses you've selected will run will run automatically.

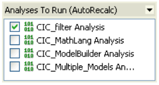
Direct entry

- Type a new value in the Value box for the variable.
- When you press the enter key it will be entered and the analyses you've selected will run.

When values have been set and you want to save a set under a new name just type the name into the name entry field and click the save settings (diskette) icon. If you want to save the current state of graphs, click the graph checkpoint button to create a named checkpoint.

Quicker Tuning: don't tune more than you need

- In the Tune Window, only enable the Analyses that you want automatically recalculated after you tune.
- Check / uncheck analyses. When checked, they will automatically recalculate (so they will run while tuning).

- 
Analyses To Run (AutoRecalc)
CIC_filter Analysis
CIC_MathLang Analysis
CIC_ModelBuilder Analysis
CIC_Multiple_Models An...

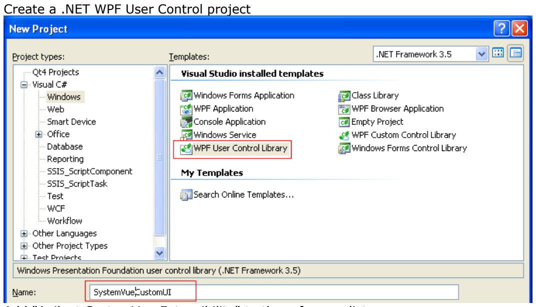
UI Customizations

Contents

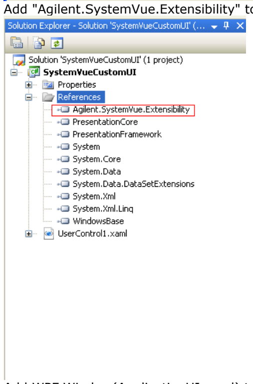
Add Customized UI for Applications

Steps to create a customized UI for part FFT_Cx in Algorithm Design library.

1. Create a .NET WPF User Control project



2. Add "Agilent.SystemVue.Extensibility" to the reference list



3. Add WPF Window(ApplicationUI.xaml) to Project SystemVueCustomUI
4. Add Code to the User Control
   - Use namespace Agilent.SystemVue.Extensibility;
   - Inherit interface IApplicationCustomUI

```
    public partial class ApplicationUI : Window,IApplicationCustomUI
```

   - Define IModel instance to deal with data exchange with SystemVue

```
        private IModel model;
```

   - Implement OnConnection function

```
        public void OnConnection(IModel model)
        {
            //Step5: assign model
            this.model = model;
            try
            {
                // Step6: initialize application UI
                int val;
                this.model.GetParameter("Design1", "F1", Direction.Name, out val);
                Direction.SelectedIndex = val;

                this.model.GetParameter("Design1", "F1", FreqSequence.Name, out val);
                FreqSequence.SelectedIndex = val;
```

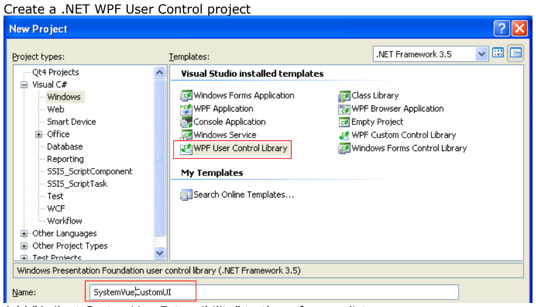
   - Add event handler to update the UI change to design/model

```
        // Step 7: update the UI to design/model
        private void Direction_SelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            this.model.SetParameter("Design1", "F1", Direction.Name, Direction.SelectedIndex);
        }
```

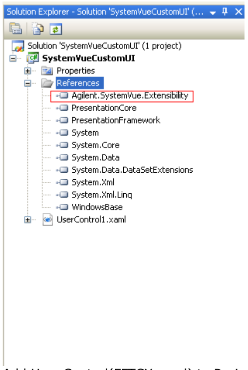
Add Customized UI for Models

Steps to create a customized UI for part FFT_Cx in Algorithm Design library.

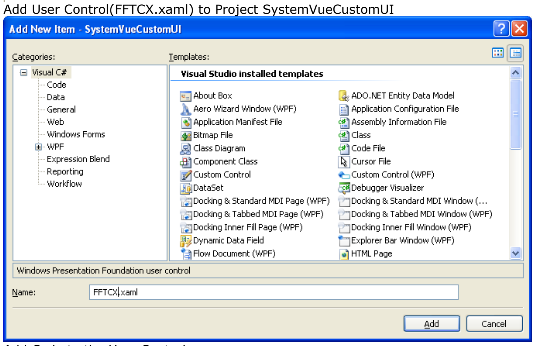
1. Create a .NET WPF User Control project



2. Add "Agilent.SystemVue.Extensibility" to the reference list



3. Add User Control(FFTCX.xaml) to Project SystemVueCustomUI



4. Add Code to the User Control
   - Specify the ModelUI Property

```
[ModelUI("FFT_Cx@Data Flow Models")]
```

   - Inherite Interface IModelCustomUI

```
public partial class FFTCX : UserControl, IModelCustomUI
```

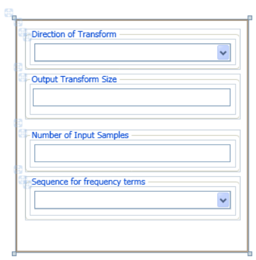
   - Define the model instance, model is used to deal with the interaction of data

```
private IModel model;
```

   - Implement OnConnection function, it will initialize model

```
public void OnConnection(IModel model)
{
    this.model = model;
}
```

   - Design Control Panel




```
<UserControl x:Class="SystemVueCustomUI.FFTCX"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="300" Width="300">
    <Grid>
        <GroupBox Header="Direction of Transform" Margin="12,12,11,0" Name="groupBox1" Height="52" VerticalAlignm
            <Grid>
                <ComboBox Margin="6,6,6,0" Name="Direction" Height="23" VerticalAlignment="Top" SelectionChanged=
```

> If UI control's Name is the same as parameter's name, UI control will be initialized by SystemVue

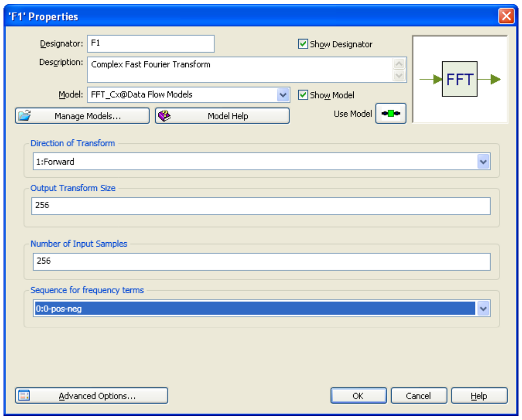
   - Add event handler to the control in order to set the parameter to the model

```
private void Direction_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    this.model.SetParameter(Direction.Name, Direction.SelectedIndex);
}
```

5. How does SystemVue find the customized UI for models?
   - SystemVue will search for the customized UI in folder "My Documents\Agilent\SystemVue\CustomUIs".
   - SystemVue will search for the customized UI in customer model library folder.

Introduction

Custom UI is a way to provide a control panel for SystemVue applications or SystemVue models.
For example:



Please note that control panel only supports .NET WPF Page or UserControl as the base class

Agilent.SystemVue.Extensibility

Assembly Agilent.SystemVue.Extensibility provides several application interfaces to get/set parameter values for model and design.

1. Interface **IModel**
   Methods:

Name	Description	Model/Application
void GetParameter(string name, out string value)	Get parameter as a string value	Model
void SetParameter(string name, string value)	Set a string to parameter	Model
void GetParameter(string name, out int value)	Get parameter as a int value	Model
void SetParameter(string name, int value)	Set a int value to parameter	Model
void GetParameter(string name, out bool value)	Get parameter as a boolean value	Model
void SetParameter(string name, bool value)	Set a boolean value to parameter	Model
void GetParameter(object rb)	Get parameter as .NET object	Model
void SetParameter(object rb)	Set a .NET object to parameter	Model
void GetParameter(string designName, string modelName, string param, out string value)	Get a design/model's parameter value	Application
void SetParameter(string designName, string modelName, string param, string value)	Get a design/model's parameter value	Application
void GetParameter(string designName, string modelName, string param, out int value)	Get a design/model's parameter value	Application
void SetParameter(string designName, string modelName, string param, int value)	Get a design/model's parameter value	Application
void SelectSchematicAndRun(string designName)	Run the specified design	Application
void SelectEquationAndCalculate(string equationName)	Run the specified equation	Application

2. Interface **IApplicationCustomUI**

Name	Description
void OnConnection(IModel model);	As a callback function to pass IModel to customized UI

3. Interface **IModelCustomUI**

Name	Description
void OnConnection(IModel model);	As a callback function to pass IModel to customized UI

User Defined Models

Contents

Catapult C Flow

Catapult C Synthesis is an algorithmic synthesis tool by Mentor Graphics that synthesizes C++ code into RTL (VHDL, Verilog, and SystemC). SystemVue provides a design flow that can be integrated inside Catapult System Level Synthesis tool to automatically generate SystemVue C++ model from user's C++ input to Catapult. The resulting SystemVue C++ model can be compiled using supported version of Visual Studio and then resulting dll can be loaded inside SystemVue to use the model, please see *Loading and Debugging a C++ Model Library* (users).

Configuring Catapult to Use SystemVue Flow

In order to use SystemVue flow inside Catapult, you need to add path to SystemVue flow directory in **Flow Search Path** in Catapult. This can be done as follows.

1. Start Catapult System Level Synthesis Tool.
2. Click on Tools > Set Options... > Flows > Flow Search Path
3. Add **<SystemVue Installation Directory>\ModelBuilder\Catapult** to the Flow Search Path, where <SystemVue Installation Directory> is the directory where you have installed SystemVue.
4. Click **Apply and Save** to save this information in Catapult registry.
5. Either start a new project or restart Catapult to load SystemVue flow.

Using SystemVue Flow

To use SystemVue flow, enable it in the Flow Manager window of Catapult. The SystemVue flow is enabled for "extract" stage in Catapult. Once the flow is enabled and design has entered "extract" stage, the SystemVue flow will generate a C++ SystemVue model around the top-level function, which was synthesized inside Catapult. If supported version of Visual Studio is installed, the flow will also create a Visual Studio project with correct configuration and opens it automatically. By default, the generated files, and Visual Studio project are stored in SystemVue sub-directory inside the solution directory, to override this behavior specify a directory in **SystemVue model directory** option of SystemVue flow. You can compile the Visual Studio project and then load the resulting dll inside SystemVue to use the model.

Understanding Generated SystemVue Model

The generated SystemVue model will have only Fixed Point (AgilentEESof::FixedPoint) type ports with input/output port directions based on the synthesized port directions in Catapult. The directions are inferred as follows

Catapult Direction	SystemVue Direction
IN	input
OUT	output
INOUT	1 input and 1 output

For INOUT type port, 1 input and 1 output is generated in SytemVue model. The model reads all the inputs in SystemVue and then call the top-level function to calculate the outputs. The SystemVue flow currently supports following data types as top level function parameter and return type, and then infer the SystemVue Fixed Point port precision as shown in the following table

Supported Data Type / Class	Fixed Point Port Precision
bool	wl=1; iwl=1; sign=unsigned
char	wl=8; iwl=8; sign=signed
unsigned char	wl=8; iwl=8; sign=unsigned
short	wl=16; iwl=16; sign=signed
unsigned short	wl=16; iwl=16; sign=unsigned
int, long, signed	wl=32; iwl=32; sign=signed
unsigned int, unsigned long, unsigned	wl=32; iwl=32; sign=unsigned
long long	wl=64; iwl=64; sign=signed
unsigned long long	wl=64; iwl=64; sign=unsigned
double, float	wl=32; iwl=16; sign=signed (modify the generated code if you need a different precision)
ac_int	wl=infer from ac_int, iwl=wl, sign=infer from ac_int
ac_fixed	infer all precision options from ac_fixed

Only scalar, pointer and single dimension array version of above data-types are supported. Multi-dimensional arrays, struct, and user defined classes are not supported in SystemVue flow.

Multirate Properties

The generated port corresponding to a return value, a scalar parameter, and a pointer parameter to the top level function is always uni-rate. For example the top level function

```
void fir_filter (  ac_int<8> *input, int  gain, ac_int<8> *output);
```

will have 3 uni-rate ports.

The generated port corresponding to an array type parameter to the top level function is always multi-rate with rate equals to the number of elements in the array. For example the top level function

```
void fir_filter (ac_int<10> *input, ac_int<10> coeffs[8], ac_int<10> *output );
```

will have two uni-rate ports **input**, and **output**, and one multi-rate port **coeffs** with rate=8.

Header Files

You must include a header file in the Catapult project, which contains the top-level function signature and all #define statements needed for this function declaration.

Using Static Variables

If you use static variables inside a function/method, please make sure not to use **Multithreaded Simulation** inside SystemVue. If you must use **Multithreaded Simulation** then use of all static variables must be thread safe, either using lock(s) or using any other method. Also, the values stored in static variables are kept in subsequent simulation and are not cleared, if you want to clear these values then you must write the code accordingly and must clear the values explicitly.

Example of Generated SystemVue Model

If the top-level function declaration is

```
void fir_filter (ac_int<8> *input, ac_int<8> coeffs[NUM_TAPS], ac_int<8> *output );"
```

then following SystemVue model will be generated. Please note that port direction is inferred based on the actual C++ source code (not shown) by Catapult and the same information is used to infer SystemVue port direction as shown in the table above.

```
// -------------------------------------------------------
// file : SystemVueModel_fir_filter.h
// Class : SystemVueModel_fir_filter
// Autogenerated file from Catapult SystemVue flow 2011.03
// Date: 10/15/2010   Time: 16:46:49
// Copyright (c) 2000-2011, Agilent Technologies, Inc.
// -------------------------------------------------------
#pragma once
#include "ModelBuilder.h"
#include "DFFixedPointInterface.h"
```

```
class SystemVueModel_fir_filter :
public AgilentEEsof::DFModel, public AgilentEEsof::DFFixedPointInterface
{
public:

 // IN port representing  input
 AgilentEEsof::FixedPointCircularBuffer m_SystemVue_input;

 // IN port representing  coeffs
 AgilentEEsof::FixedPointCircularBuffer m_SystemVue_coeffs;

 // OUT port representing  output
 AgilentEEsof::FixedPointCircularBuffer m_SystemVue_output;

 // For Computation
 bool Run();

 // For multi-rate setup, FixMe: commented out untill implemented
 bool Setup();

 // It is a fixed point model, we need to propagate fixed point parameters
 ERESULT SetOutputFixedPointParameters();

 DECLARE_MODEL_INTERFACE( SystemVueModel_fir_filter );

};


// ---------------------------------------------------------
// file : SystemVueModel_fir_filter.cpp
// Class : SystemVueModel_fir_filter
// Autogenerated file from Catapult SystemVue flow 2011.03
// Date: 10/15/2010   Time: 16:46:49
// Copyright (c) 2000-2011, Agilent Technologies, Inc.
// ---------------------------------------------------------
// SystemVue model header file
#include "SystemVueModel_fir_filter.h"
// Header files from Catapult design
#include "fir_filter.h"
// Header to convert data to/from ac datatypes from/to AgilentEEsof::FixedPoint
#include "FixedPoint_AC.h"


// ---------------------------------------------------------------------------------
// DEFINE_MODEL_INTERFACE  : Defines I/O ineterface in SystemVue
// ---------------------------------------------------------------------------------
#ifndef SV_CODE_GEN
DEFINE_MODEL_INTERFACE ( SystemVueModel_fir_filter )
{
 SET_MODEL_NAME( "fir_filter" );
 SET_MODEL_CATEGORY( "Catapult" );

 // Add input representing  input
 AgilentEEsof::DFPort port_input = ADD_MODEL_INPUT( m_SystemVue_input );
 port_input.SetName("input");

 // Add input representing  coeffs
 AgilentEEsof::DFPort port_coeffs = ADD_MODEL_INPUT( m_SystemVue_coeffs );
 port_coeffs.SetName("coeffs");

 // output representing  output
 AgilentEEsof::DFPort port_output = ADD_MODEL_OUTPUT( m_SystemVue_output );
 port_output.SetName("output");


 return true;
}
#endif
// ---------------------------------------------------------------------------------
// bool Setup() : Setup multi-rate properties
// ---------------------------------------------------------------------------------
bool SystemVueModel_fir_filter::Setup ( )
{
 m_SystemVue_coeffs.SetRate(8);
 return true;
}
// ---------------------------------------------------------------------------------
// ERESULT SetOutputFixedPointParameters() : Set output ports fixed point precision
// ---------------------------------------------------------------------------------
ERESULT SystemVueModel_fir_filter::SetOutputFixedPointParameters ( )
{
 // output representing  output
 ac_int<8, true > output;
 m_SystemVue_output.SetParameters ( AgilentEEsof::GetFixedPointParameters ( output ) );
 return NOERROR_;
}

// ---------------------------------------------------------------------
// bool Run() : performs scientific calculation
// ---------------------------------------------------------------------
bool SystemVueModel_fir_filter::Run ( )
{
 ac_int<8, true > input;
 ac_int<8, true > coeffs[8];
 ac_int<8, true > output;
 // Transfer data from SystemVue inputs to function input
 AgilentEEsof::FixedPoint_To_AC( m_SystemVue_input[0], input );
 for(size_t i=0; i < 8; i++)
 {
 AgilentEEsof::FixedPoint_To_AC( m_SystemVue_coeffs[i], coeffs[i] );
 }
 // Call the C++ function
 fir_filter(&input, coeffs, &output);
 // Transfer data from function output to SystemVue output
 AgilentEEsof::AC_To_FixedPoint( output, m_SystemVue_output[0]);
 return true;
}
```

Creating a Custom C++ Model Library

This section details the step by step procedure that you need to follow to create a custom C++ Model libray. The C++ models are fullly configurable using C++ API presented in this section. You could add custom inputs, outputs, bus inputs, bus outputs, and parameters to customize your model.

Contents

Advanced Topics

Defining the Model Library Properties

To override the default properties of your C++ library, you need to define the `bool DefineLibraryProperties(AgilentEEsof::LibraryProperties* pLibraryProperties)` function in your library.

Specifying the Library Name in SystemVue

By default, the Part, Model and Enumeration libraries names will be derived from your DLL name. You can override this behavior, by calling the `LibraryProperties::SetLibraryName` method.

Removing the Model Library Path from Auto-generated Parts

By default, all Parts that are automatically created during the load of your library will reference the Models using the full path to the model library. You can override this behavior by using the `LibraryProperties::SetExcludeLibrarySuffixFromPartModels` method.

For example, in Visual Studio solution in the *Quick start* (users) section, the **AddCx** part specifies its associated model to be **AddCx@Custom Models**. If you call `LibraryProperties::SetExcludeLibrarySuffixFromPartModels` method, the model will simply be listed as **AddCx**.

To learn more about the library manager, refer to the *Using the Library Manager* (users) documentation.

Embedding your own XML Libraries into the DLL

You may create your own XML libraries using SystemVue that you can embed into the DLL. These XML libraries can then be loaded into SystemVue at the same time the DLL is loaded.

The XML file generated by SystemVue must be imported as a Resource into your Visual Studio project. During the Resource Import process, you will be prompted to provide a name for the Resource Type. You may provide any name you want, but we recommend "XML". Your imported resource will receive a corresponding ID which must be used to register the resource.

Register your XML resource by calling the `LibraryProperties::AddLibrary` method. The syntax for the method is as follows:

AddLibrary(iResourceID, strResourceType, bMerge)

where iResourceID is the resource ID of the imported XML resource, strResourceType is the name of the resource type that was given when the resource was imported, and bMerge specifies whether or not the library should be merged with the auto-generated library of the same type - explained in further detail below.

When a DLL is loaded, certain libraries are automatically generated. For example, a Design library consisting of generated model templates for each model defined in the DLL is produced. A Part library is generated as well, unless you provide your own and mark all of the models as not needing an auto-generated part (done with the DISABLE_PART_GENERATION() macro in your DEFINE_MODEL_INTERFACE function).

Example

```
#include "Stdafx.h"
#include "LibraryProperties.h"
bool DefineLibraryProperties(AgilentEEsof::LibraryProperties* pLibraryProperties)
{
  // Define the library name for the Part, Model and Enum libraries.
  // By default, the DLL name is used.
  pLibraryProperties->SetLibraryName("C++ Model Builder");
  // Strip the @Library suffix in the model name for auto-generated parts.
  // This allows you to use the Library Manager search path to easily override the model
  // by changing the search path.  By default, all generated parts
  // will reference the models using the full path.
  pLibraryProperties->SetExcludeLibrarySuffixFromPartModels();
  // Declare that we have an embedded XML library that we added as a resource.
  // The library can be of any object type but in this case we have a Part library and
  // a Symbol (Design) library.
  // We imported the XML libraries as resources of type "XML", and we would like
  // the Parts in this library to be merged into the library of auto-generated parts
  // that is created for this library.
  // However, we want the symbol library to be imported as-is.
  pLibraryProperties->AddLibrary( ID_XML_PARTS, "XML", true );
  pLibraryProperties->AddLibrary( ID_XML_SYMBOLS, "XML", false );
  // Return true to indicate success.  If you return false, {PRODUCT-NAME}
  // will report that it was unable to load the library.
  return true;
}
```

Supporting standalone use of DFModels

To enable use of you models in standalone programs (including those generated by SystemVue), you should check the `SV_CODE_GEN` c preprocessor definition as shown below:

```
#ifndef SV_CODE_GEN
DEFINE_MODEL_INTERFACE(Adder)
{
 ADD_MODEL_INPUT(In1);
 ADD_MODEL_INPUT(In2);
 ADD_MODEL_OUTPUT(Out);
 ADD_MODEL_PARAM(Gain);
 return true;
}
#endif
```

Object Composition

At times, you may find it useful to develop a set of C++ classes to define a common set of ports, parameters or algorithms for use by your custom {{DFModel}}s. In object oriented design, this is called *object composition* or defining a *has a* relationship to another class. To learn more about object composition, refer to the Wikipedia article .

If you use object composition to define ports or parameters, you should use the `DFPort::PrependCodeGenName(const char* pccCodeGenPath)` and `DFParam::PrependCodeGenName(const char* pccCodeGenPath)` methods to aid in code generation. These methods prepend a data member name to the code generation name of the port or parameter. A null pointer argument to these methods is ignored. The path should include '.' or '->' to define the access to the data members.

Below is a example class - in the example below, we define a `AddInterface` method to assist in object composition using this class.

```
class WriteData
{
public:
 int Start; // Index to start writing data
 int Stop;  // Index to stop writing data
 void AddInterface(AgilentEEsof::DFInterface& model, const char* pccCodeGenPath)
 {
 AgilentEEsof::DFParam start = ADD_MODEL_PARAM(Start);
 start.SetDefaultValue("0");
 start.PrependCodeGenName( pccCodeGenPath);
 AgilentEEsof::DFParam stop = ADD_MODEL_PARAM(Stop);
 stop.SetDefaultValue("100");
 stop.PrependCodeGenName( pccCodeGenPath);
 }
```

```
 void Initialize()
 {
 m_iCount = 0;
 }
       bool WriteSample(double d)
 {
 if ( m_iCount <= Stop)
 {
  // Write data d
  m_iCount++;
  return true;
 }
 else
 {
  return false;
 }
 }
private:
 int m_iCount;
};
```

A `DFModel` could use this class by object composition.

```
// Code for header file
class CustomModel : public AgilentEEsof::DFModel
{
public:
 DECLARE_MODEL_INTERFACE(CustomModel);
 bool Initialize();
 bool Run();
 WriteData m_WriteData;

 double Input;
};
// Code for source file
DEFINE_MODEL_INTERFACE(CustomModel)
{
 // Notice the name to be prepended includes both the name of the data member
 // and a '.'. Had this been a pointer to WriteData, the argument would
 // have been "m_WriteData->".
 m_WriteData.AddInterface(model, "m_WriteData.");
 // Add a input port
 ADD_MODEL_INPUT(Input);
 return true;
}
bool CustomModel::Initialize()
{
 m_WriteData.Initialize();
 return true;
}
bool CustomModel::Run()
{
 m_WriteData.WriteSample(Input);
 return true;
}
```

Writing C++ Models for Code Generation

In general, SystemVue *C++ code generator* (users) supports any C++ model that is created and loaded based on *Creating a Custom C++ Model Library* (users). However, in order to successfully compile generated code, additional information needs to be provided in *DEFINE_MODEL_INTERFACE* (users) of C++ models that are going to be used in code generation. For more detail, please refer to *Writing C++ Models for Code Generation* (users).

Using Third Party Library in C++ Models

This section provides a general guideline to use third party library content in custom C++ models.

1. Follow the steps in *Writing C++ Models* (users) and *Building C++ Model Library* (users). Include the third party header files in the custom .h and .cpp files.
2. In the project property page (in Solution Explorer, right click the project, then choose Properties), set the third party include directories in Configuration Properties > C/C++ > General > Additional Include Directories.
3. In the project property page, set the third party library directories in Configuration Properties > Linker > General > Additional Library Directories.
4. In the project property page, set the third party .lib files in Configuration Properties > Linker > Input > Additional Dependencies.
5. If the third party library is built as dynamic link library (DLL), you must set windows **PATH** environment variable to point to the directory where third party DLL is located before starting SystemVue or any executable using C++ Model.

Windows search for PATH environment variable when loading a DLL, not specifying the directory containing the DLL in PATH may result in un-predictable behavior including crash.

Example – Using Matlab Compiled Libraries in C++ Models

This example introduces how to compile a Matlab function into dynamic link library and use it in SystemVue C++ Model.

This Matlab code implements a feed-forward equalizer (FFE) function. Suppose it is written in a file called "MyFFE.m".

```
% MyFFE.m
function [ out ] = MyFFE( Coefficients, SamplesPerBit, Reset, in )
% Declare persistent in order to preserve internal state
persistent dSamples;
persistent numSamples;
persistent taps;
if ( isempty(dSamples) || Reset )
numSamples = length(Coefficients) * SamplesPerBit;
dSamples = zeros(1, numSamples);
taps = Coefficients';
out = 0.0;
end
if ( ~Reset )
dSamples = [in,dSamples(1:numSamples-1)];
out = dSamples(1:SamplesPerBit:numSamples) * taps;
end
end
```

Users can declare **persistent** variables in Matlab function to preserve internal state.

By using the following Matlab command, Matlab compiler compiles "MyFFE.m" into "libmyffe.h", "libmyffe.lib", "libmyffe.dll", as well as other relevant files. Please refer to Matlab document for more details.

```
mcc -W cpplib:libmyffe -T link:lib  MyFFE.m
```

The following code segment is generated by Matlab compiler as part of "libmyffe.h", which declares **MyFFE** function in C++. MyFFE function is the entry point that performs compiled FFE operation in the library ("libmyffe.dll").

```
extern LIB_libmyffe_CPP_API void MW_CALL_CONV MyFFE(int nargout, mwArray& out, const mwArray& Coeff
icients, const mwArray& SamplesPerBit, const mwArray& in);
```

The following code segment is also generated by Matlab compiler as part of "libmyffe.h", which initializes and terminates the library respectively.

```
extern LIB_libmyffe_C_API
bool MW_CALL_CONV libmyffeInitialize(void);
extern LIB_libmyffe_C_API
void MW_CALL_CONV libmyffeTerminate(void);
```

The following code shows how to write a SystemVue C++ model,

Compiled_M_Code_FFE, that uses Matlab-generated **MyFFE** function to perform FFE operation.

```
//Compiled_M_Code_FFE.h
#pragma once
#include "ModelBuilder.h"
class mwArray;
class Compiled_M_Code_FFE : public AgilentEEsof::DFModel
{
public:
 Compiled_M_Code_FFE();
 ~Compiled_M_Code_FFE();
 bool Initialize();
 bool Run();
 bool Finalize();
 // parameters
 double *m_dCoefs;
 int m_iCoefsSize;
 int m_iSamplesPerBit;
 // i/o
 double m_dInput, m_dOutput;
 DECLARE_MODEL_INTERFACE(Compiled_M_Code_FFE);
private:
 mwArray *m_pMatlabInput, *m_pMatlabOutput;
 mwArray *m_pMatlabCoefs, *m_pMatlabSamplesPerBit, *m_pMatlabReset;
 bool m_bReset;
};
```

```
//Compiled_M_Code_FFE.cpp
#include "Compiled_M_Code_FFE.h"
#include "../libmyffe.h"
#ifndef SV_CODE_GEN
DEFINE_MODEL_INTERFACE(Compiled_M_Code_FFE)
{
 SET_MODEL_DESCRIPTION("Feed-Forward Equalizer");
 ADD_MODEL_HEADER_FILE("Compiled_M_Code_FFE.h");
 SET_MODEL_CATEGORY("IBIS-AMI Transceivers");
 AgilentEEsof::DFPort port = ADD_MODEL_INPUT(m_dInput);
 port.SetName("input");
 port = ADD_MODEL_OUTPUT(m_dOutput);
 port.SetName("output");
 AgilentEEsof::DFParam param = ADD_MODEL_ARRAY_PARAM(m_dCoefs,m_iCoefsSize);
 param.SetName("Coefficients");
 param.SetDefaultValue("[1  0.1 0.2]");
 param.SetDescription("Bit level FFE taps");
 param = ADD_MODEL_PARAM(m_iSamplesPerBit);
 param.SetName("SamplesPerBit");
 return true;
}
#endif
Compiled_M_Code_FFE::Compiled_M_Code_FFE() : m_pMatlabInput(0), m_pMatlabOutput(0), m_pMatlabCoefs(0
), m_pMatlabSamplesPerBit(0), m_pMatlabReset(0)
{
 m_iSamplesPerBit = 16; // default value
}
Compiled_M_Code_FFE::~Compiled_M_Code_FFE()
{
}
bool Compiled_M_Code_FFE::Initialize()
{
 bool bStatus = true;
 libmyffeInitialize();  // Initialize library
 m_pMatlabInput = new mwArray(1,1,mxDOUBLE_CLASS);
 m_pMatlabOutput = new mwArray(1,1,mxDOUBLE_CLASS);
 m_pMatlabCoefs = new mwArray(1,m_iCoefsSize,mxDOUBLE_CLASS);
 m_pMatlabCoefs->SetData(m_dCoefs,m_iCoefsSize );
 m_pMatlabSamplesPerBit = new mwArray(m_iSamplesPerBit);
 // Reset coeffs, samples per bit, and internal buffer holding input signal values
 m_bReset = true;
 m_pMatlabReset = new mwArray(m_bReset);
 MyFFE(1,*m_pMatlabOutput,*m_pMatlabCoefs,*m_pMatlabSamplesPerBit,*m_pMatlabReset,*m_pMatlabInput
);
 delete m_pMatlabReset;
 m_bReset = false;
 m_pMatlabReset = new mwArray(m_bReset);
 return bStatus;
}
bool Compiled_M_Code_FFE::Run()
{
 bool bStatus = true;
 m_pMatlabInput->SetData(&m_dInput,1);
 MyFFE(1,*m_pMatlabOutput,*m_pMatlabCoefs,*m_pMatlabSamplesPerBit,*m_pMatlabReset,*m_pMatlabInput
); // Call MyFFE
 m_pMatlabOutput->GetData(&m_dOutput,1);
 return bStatus;
}
bool Compiled_M_Code_FFE::Finalize()
{
 bool bStatus = true;
 libmyffeTerminate();  // Terminate library
 delete m_pMatlabInput;
 delete m_pMatlabOutput;
 delete m_pMatlabCoefs;
 delete m_pMatlabSamplesPerBit;
 delete m_pMatlabReset;
 m_pMatlabSamplesPerBit = 0;
 m_pMatlabCoefs = 0;
 m_pMatlabInput = 0;
 m_pMatlabOutput = 0;
 m_pMatlabReset = 0;
 return bStatus;
}
```

Call Matlab-generated initialize function, e.g., libmyffeInitialize(), in the Initialize() method of the SystemVue model to properly initialize the library. Also call Matlab-generated terminate function, e.g., libmyffeTerminate(), in the Finalize() method of the SystemVue Model to properly close the library.

The following steps guide users to setup a Visual Studio project to build Compiled_M_Code_FFE into SystemVue model library.

1. Follow these steps to *setup a Visual Studio project* (users). Suppose the project name is "SystemVue Compiled M Code Models".
2. Copy "libmyffe.h" and "libmyffe.lib" to the project directory.
3. Add "libmyffe.h" into the project Header Files.
4. Create "Compiled_M_Code_FFE.h" and "Compiled_M_Code_FFE.cpp" as shown above into the project.
5. Suppose the Matlab installation directory is "C:\Program Files\MATLAB\R2010a". Add "C:\Program Files\MATLAB\R2010a\extern\include" in Configuration Properties > C/C++ > General > Additional Include Directories.
6. Add "..";"C:\Program Files\MATLAB\R2010a\extern\lib\win32\microsoft" in Configuration Properties > Linker > Input > Additional Dependencies. The path ".." refers to the location containing "libmyffe.lib".
7. Add "libmyffe.lib" and "mclmcrrt.lib" in Configuration Properties > Linker > Input > Additional Dependencies.
8. Build the solution. The resulting "SystemVue Compiled M Code Models.dll" is a custom SystemVue library that contains Compiled_M_Code_FFE model.
9. Set windows **PATH** environment variable to include the directory where "libmyffe.dll" is located. You must do it before starting SystemVue.

If there is a persistent variable in the Matlab code, only one instance of such SystemVue model can be placed on the schematic. If there are multiple SystemVue model instances that invoke the same Matlab function, such persistent variable will be shared by multiple instances, and may cause unexpected simulation results.

Since both SystemVue and Matlab use Intel Math Kernel Library (MKL), you need to set an environment variable "KMP_DUPLICATE_LIB_OK=TRUE".

To load and run the DLL, you will need to add the MATLAB Compiler ™ Runtime v710 (or v711) (or v711) Win32 DLLs to your PATH environment variable. The default install location on a Win32 PC is: "C:\Program Files\MATLAB\MATLAB Compiler Runtime\v710\bin\win32". These DLLs are available in the MATLAB Compiler ™ Runtime v710 installers from the MathWorks™.

Writing Data Flow C++ Models

If you have not done so yet, please read and understand the *Building your first C++ Model Library* (users) section, especially for understanding on how to *create/setup Visual Studio project* (users) for Data Flow C++ models. The rest of this section assumes that you are familiar with setting up a Visual Studio project for Data Flow C++ models.

After you have *setup the Visual Studio project* (users), you can write a C++ class that represents a Data Flow model. All C++ Data Flow models must be written as a C++ class. The C++ class for the model must be derived from **AgilentEEsof::DFModel** with **public** access. Each model requires one class, and each class can support only one model. It is not possible to write multiple models inside a single C++ class. Inheritance and *Object Composition* (users) is permitted, to avoid code duplication.

Writing Header file for the C++ Class

Follow the instructions below to declare your C++ class in the header.

- You must include the header **ModelBuilder.h**.
- The C++ class must be derived from **AgilentEEsof::DFModel** with **public** access.
- All data members which will act as input/output ports or parameters must have **public** access. The supported data types for inputs/outputs and parameters are listed in the sections *Data Types Used as Inputs/Outputs* (users), and *Data Types Used as Parameters* (users) respectively in *Supported Data Types* (users) page.
- You must call the macro **DECLARE_MODEL_INTERFACE(<class name>);** in the **public** section of your class declaration, where <class name> is the name of your model class. This step is very important as it declares an interface between the C++ model class and the simulator.
- You must override **bool Run()** method of the **AgilentEEsof::DFModel** with public access. The **Run()** method has special meaning for the simulator which will be discussed later in this document.
- Optionally you can also override **bool Setup()**, **bool Initialize()**, and **bool Finalize()** methods with public access. These methods have special meaning for the simulator which will be discussed later in the document.
- You can add any other method(s) that is needed for your model implementation or add non input/output/parameter data members which are needed for your model implementation with private, protected or public access of your choice.
- For a simple example header file please read **Adder.h** file in the section *Adding a new Model to the Project* (users) in First Custom C++ Model Library section.

Writing cpp file for the C++ Class

Steps for writing a cpp file for the C++ Class are detailed below:

- Defining the interface using DEFINE_MODEL_INTERFACE macro
- Defining the Run() method.

For a simple example cpp file please read **Adder.cpp** file in the section *Adding a new Model to the Project* (users) in First Custom C++ Model Library section.

Defining Interface to the Simulator

The interface to the simulator i.e. inputs, outputs and parameters must be defined inside the mandatory **DEFINE_MODEL_INTERFACE(<class name>) { }** macro, where <class name> is the name of the model class. The DEFINE_MODEL_INTERFACE macro must return true on success and false on failure.

Adding Inputs to the Interface

- The **ADD_MODEL_INPUT(<data member>)** macro can be used to add a class data member declared with public access to the interface as an input port.
- The data types supported by **ADD_MODEL_INPUT** macro are listed at *Data Types Used as Inputs/Outputs* (users).
  - The *C++ Built In Data Types as Uni-rate Inputs/Outputs* (users) are added as uni-rate inputs. The uni-rate inputs are those inputs which consume one data point for each invocation of the model by the simulator.
  - The *C++ Built In Pointer Data Types as Multi-rate Inputs/Outputs* (users) are added as multi-rate inputs. The multi-rate inputs are those inputs which can consume one or more data points for each invocation of the model by the simulator.
  - The *SystemVue CircularBuffer Data Types* (users) are added as multi-rate inputs.
- The simulator is responsible for allocating and releasing memory for pointer data members, and for CircularBuffer data types.
- The default rate of a multi-rate port is "1" which could be changed by adding a rate-variable for pointer data members and setting the value of this rate-variable in **bool Setup()** method, or by calling the SetRate() method on an object of a CircularBuffer type inside **bool Setup()**.
- By default, the name of the input port is chosen to be the name of the data member added using the **ADD_MODEL_INPUT** macro.
- The **ADD_MODEL_INPUT** macro returns an object of type **AgilentEEsof::DFPort** that you could further use only inside the **DEFINE_MODEL_INTERFACE** to add a rate-variable for multi-rate ports or to change the port-name.
- The details of using **AgilentEEsof::DFPort** are discussed later in the document.

Adding Outputs to the Interface

- The **ADD_MODEL_OUTPUT(<data_member>)** macro can be used to add a class data member declared with public access to the interface as an output port.
- The **ADD_MODEL_OUTPUT** macro is similar to ADD_MODEL_INPUT macro except that it adds an output to the interface instead of input.

Adding Parameters to the Interface

- The **ADD_MODEL_PARAM(<parameter_data_member>)** can be used to add a *C++ Built In Scalar Data Types* (users), supported *AgilentEEsof::Matrix* (users) or *SystemVue Built In Enumerations* (users) as a parameter.
- The **ADD_MODEL_ARRAY_PARAM(<parameter_data_member>,<parameter_array_size_data_member>)** can be used to add *C++ Built In Pointer Data Types* (users) as an array type parameter.
- The **ADD_MODEL_ENUM_PARAM(parameter_data_member, enum_type_name)** can be used to add a *User Defined Enumeration* (users) as enumerated parameter using a user-defined C++ **enum**.
  Where;
- The <parameter_data_member> is the class data member that will be set by the simulator to hold the parameter value.
- The <parameter_array_size_data_member> is the class data member that will be set by the simulator to hold the number of elements present in an array type parameter. The data type of <parameter_array_size_data_member> must be **unsigned**.
- The <enum_type_name> is the name of enum type used to instantiate corresponding <parameter_data_member>.
- The value of parameters and number of elements in case of array type parameters will be available to be read in Setup(), Initialize(), Run() and Finalize() methods.
- The simulator is responsible to allocate/release all memories and setting up the parameter values.
- The *ADD_MODEL_PARAM*, *ADD_MODEL_ARRAY_PARAM* , and *ADD_MODEL_ENUM_PARAM* macros return an object of type **AgilentEEsof::DFParam** that you could further use only inside the *DEFINE_MODEL_INTERFACE* to change the parameter name, description, to set a char * type parameter as a file type parameter, to add possible enumerations for an enumerated parameter, and to convert an integer type variable to use one of the

predefined enumeration. The details of using **AgilentEEsof::DFParam** are discussed later in the document.

- Some special considerations are required when using the **ADD_MODEL_ENUM_PARAM** to add *User Defined Enumeration* (users). After adding the data member of user defined enumration type, we need to explicitly add the specific enumerations to the parameter. This could be done using **AddEnumeration(const char * name, int value)** method of **AgilentEEsof::DFParam** object returned by *ADD_MODEL_ENUM_PARAM*. As an example if our Adder example above can take only selected values for its Gain parameter then we can modify the header and cpp files as follows.

```
#pragma once
#include "ModelBuilder.h"
class Adder :
 public AgilentEEsof::DFModel
{
       enum SelectedGains {Zero, One , Three=3, Five=5};
public:
 double In1, In2; // Inputs
 double Out; // Outputs
 SelectedGains Gain; // Enumerated Parameter
 DECLARE_MODEL_INTERFACE(Adder); // Declare modelbuilder interface
 virtual bool Run();   // Method for scientific code during each invocation of the model
 virtual bool Initialize(); // This method is invoked before the start of simulation
};

#include "Adder.h"
DEFINE_MODEL_INTERFACE(Adder)
{
 ADD_MODEL_INPUT(In1);
 ADD_MODEL_INPUT(In2);
 ADD_MODEL_OUTPUT(Out);
       AgilentEEsof::DFParam enumParam = ADD_MODEL_ENUM_PARAM(Gain,SelectedGains);
       Gain  = One; //default value
       enumParam.AddEnumeration("Zero Gain",One);
       enumParam.AddEnumeration("Gain of One",One);
       enumParam.AddEnumeration("Gain of Three",Three);
       enumParam.AddEnumeration("Gain of Five",Five);
 return true;
}
bool Adder::Initialize()
{
 if (Gain == Zero)
 {
 POST_ERROR("The value of Gain cannot be == Zero Gain");
 return false;
 }
 return true;
}
bool Adder::Run()
{
 Out = Gain * (In1 + In2);
 return true;
}
```

> **Warning**
> - You must use a different parameter_data_member, and/or parameter_array_size_data_member for each parameter. No two parameters should be added with the same data member.
> - The value(s) of data members holding parameter_data_member, and parameter_array_size_data_member must not be modified by the model, these values are only modified by the simulator. Modifying these values inside the model can cause undefined behavior.
> - The **name** in **AddEnumeration(const char * name, int value)** must not contain quotation **\"** character. Failure to do this will result in not adding enumerations.

Modifying Port Properties

- The **ADD_MODEL_INPUT** and **ADD_MODEL_OUTPUT** macros return an object of type **AgilentEEsof::DFPort**. Optionally, this object can be used only inside **DEFINE_MODEL_INTERFACE** to modify the default name of the port and to add a rate-variable to the port using *C++ Built In Pointer Data Types as Multi-rate Inputs/Outputs* (users). For example to add a multi-rate input port with data member defined as **double * In**, and multi-rate output port **double * Out** and then to add a data member **unsigned uRate** to these ports, we could use the following code segment

```
// This will add a multi-rate input port with name "In" with default  rate of 1
AgilentEEsof::DFPort Inport = ADD_MODEL_INPUT(In);
// This will change the port name from "In" to "input"
Inport.SetName("input");
// Now, by changing the value of uRate in Setup() method we could change the port rate
Inport.AddRateVariable(uRate);
// This will add a multi-rate output port with name "Out" with default  rate of 1
AgilentEEsof::DFPort Outport = ADD_MODEL_OUTPUT(Out);
// This will change the port name from "Out" to "output"
Outport.SetName("output");
// Now, by changing the value of uRate in Setup() method we could change the port rate
Outport.AddRateVariable(uRate);
```

- The same data member, such as uRate in the code above, could be used to define rates for several input or output ports simultaneously. Alternatively, you could use separate data member of type unsigned for each multi-rate input or output port.
- The

```
void SetOptional(bool bIsOptional = true);
```

can be used to set an **input port** as optional port if the corresponding data member is of *SystemVue CircularBuffer Data Type* (users) . An optional input port is the port that is not required to be connceted on the schematic. To check that an input port is connected use **bool IsConnectde()** method of *SystemVue CircularBuffer Data Type* (users).

> **Warning**
> - An object of type AgilentEEsof::DFPort must only be used inside **DEFINE_MODEL_INTERFACE**. It is not legal to change port properties outside **DEFINE_MODEL_INTERFACE**.
> - The value of data member representing a port rate (uRate in the code above) must only be modified in Setup() method. Modifying the value outside the Setup() method could cause undefined behavior.
> - The rate of a *SystemVue CircularBuffer Data Type* (users) must be modified using **SetRate(uRate)** method of the corresponding object inside the Setup().
> - A *SystemVue CircularBuffer Data Type* (users) must be tested using **bool IsConnected()** before accessing if it is set as an **optional** port.

Modifying Parameter Properties

- The **ADD_MODEL_PARAM** and **ADD_MODEL_ARRAY_PARAM** macros return an object of type **AgilentEEsof::DFParam**. Optionally, this object can be used only inside **DEFINE_MODEL_INTERFACE** to change parameter name, its description, set a parameter as file, to assign a default value to the parameter, and to add an enumeration for a *User Defined Enumerated* (users) parameter. For example to add two data members **double * taps** and **int decimation** as an array and a scalar parameters with default values we could use the following code.

```
// The unsigned tapsSize will be set by simulator to the number of elements in the array
param
AgilentEEsof::DFParam paramTaps = ADD_MODEL_ARRAY_PARAM( taps, tapsSize);
paramTaps.SetDescription( "Filter tap values" );
paramTaps.SetName("FilterTaps");
// The SetDefault value method takes const char * as input, the value of this should be same
as you would enter in SystemVue
paramTaps.SetDefaultValue( "[-0.040609, -0.001628, 0.17853, 0.37665, 0.37665, 0.17853, -
0.001628, -0.040609]" );
decimation = 1; // For scalar (non-pointer) data members default can also be set before
```

```
adding as parameter
AgilentEEsof::DFParam paramDecimation = ADD_MODEL_PARAM( decimation );
paramDecimation.SetName("Decimation");
paramDecimation.SetDescription( "Decimation ratio" );
```

- The SetDefault value method takes const char * as input, the value of this should be same as you would enter in SystemVue, for enumerated parameters enter corresponding integer equivalent. For non-pointer data members, you may set the value of data member before adding it as a parameter, in this case, there is no need to use SetDefault explicitly and default is selected based on current value of the corresponding data member added as a parameter. However, for pointer type data members SetDefault must be called.
- The values of the parameters and corresponding value of data member holding array size are available to be read in the Setup(), Initialize(), Run(), and Finalize() methods. The values of array type parameters could be accessed using [] operator. The maximum index that you could use in [] operator should be less than the array parameter size set by the simulator. In the above code the maximum index for **taps** should be **tapsSize-1**.
- If you have added a parameter with data type **char *** then you can call SetParamAsFile() method of AgilentEEsof::DFParam to set it as a file parameter. If a char * is set as a file parameter then Browse button is enabled.
- If you have added a parameter as an enumeration using *ADD_MODEL_ENUM_PARAM* then **AddEnumeration(const char * EnumName, int EnumValue)** method of AgilentEEsof::DFParam must be called to add enumeration values to the simulator GUI. The *AddEnumeration* method, optionally, can be used with integer parameters as well to use integer parameter as enumeration in the simulator GUI.
- If you have added an integer parameter, you can also use **SetEnumeration(const char * EnumerationName)** method of AgilentEEsof::DFParam to use predefined enumerations. The list of predefined enumerations is mentioned in Adding Parameters to the Interface section above. The *SetEnumeration* method only supports one of the following as its parameter
  - *AgilentEEsof::QUERY_ENUM* (The possible values are QUERY_NO=0, and QUERY_YES=1)
  - *AgilentEEsof::SWITCH_ENUM* (The possible values are SWITCH_OFF=0, and SWITCH_ON=1)
  - *AgilentEEsof::BOOLEAN_ENUM* (The possible values are BOOLEAN_FALSE=0, and BOOLEAN_TRUE=1)
- For example if enumParam is an object of type AgilentEEsof::DFParam having an integer parameter then it can be used as *enumParam.SetEnumeration(AgilentEEsof::QUERY_ENUM)* to use the predefined enumeration *AgilentEEsof::QueryEnum*.
- The **SetHideCondition(const char * pcHideCondition)** method of a AgilentEEsof::DFParam object can be used to hide a parameter from GUI based on the value of another parameter in the same model. The input **pcHideCondition** to this method must be a valid MathLang conditional statement using relational operators returning true or false. The statement must use one of the parameter "names" other than the one for which the hide condition is being set. The parameter name(s) used in hide condition must be the same as set by using SetName method of a DFParam object e.g myParam.SetHideCondition("ShowAdvancedParams ~= 1"); Any parameter used in hide condition must have a name that could be used as a valid MathLang variable. Also enumeration cannot be used as is in the condition e.g myParam.SetHideCondition("ShowAdvancedParams ~= YES"); is incorrect, use myParam.SetHideCondition("ShowAdvancedParams ~= 1"); instead, if YES is equal to 1 in your enumeration list.
- The **SetSchematicDisplay(bool bDisplay)** method of a AgilentEEsof::DFParam object can be used to turn on and off the visibility of a parameter on the schematic. By default, all parameters are shown on the schematic.
- The **SetUnit(Units::UnitType eUnitType)** method of a AgilentEEsof::DFParam object can be used to set the unit of a parameter. By default, the unit is set to AgilentEEsof::Units::NONE. The supported units are
  - AgilentEEsof::Units::NONE
  - AgilentEEsof::Units::ANGLE
  - AgilentEEsof::Units::LENGTH
  - AgilentEEsof::Units::TIME
  - AgilentEEsof::Units::FREQUENCY
  - AgilentEEsof::Units::VOLTAGE
  - AgilentEEsof::Units::POWER
  - AgilentEEsof::Units::RESISTANCE
  - AgilentEEsof::Units::TEMPERATURE

Warning

- An object of type AgilentEEsof::DFParam must only be used inside **DEFINE_MODEL_INTERFACE**. It is not legal to change parameter properties outside **DEFINE_MODEL_INTERFACE**
- The value of data member added as parameter and it corresponding data member holding array size must not be modified by the model at all. These are set by the simulator automatically.

Modifying Model Properties

- By default a model is added with a default name that is same as the model class name, with no description, no category in part selector, and with an auto-generated symbol. Optionally, this default behavior can be changed, only inside **DEFINE_MODEL_INTERFACE**, using any of the four macros as shown in the following example code segment.

```
SET_MODEL_NAME("FIR");
SET_MODEL_DESCRIPTION("My First FIR filter model");
SET_MODEL_CATEGORY("DSP Filters");
SET_MODEL_SYMBOL( "SYM_FIR" );
```

Warning
- If you have decided to choose a pre-created symbol than input and output port names in your model must match those of the symbol.
- The model properties can only be changed inside **DEFINE_MODEL_INTERFACE**.

Adding Parent Model Interface

If the model is derived from an existing C++ model then the interface of the parent model could be added using th macro **ADD_PARENT_MODEL_INTERFACE(<ParentModelClass>)**, where <ParentModelClass> is the name of parent model class. If this macro is not used then, optionally, the derived model can add the complete interface itself.

The Setup() Method

Optionally the C++ model can override the **virtual bool Setup()** method of its base class **AgilentEEsof::DFModel** only to set rate variables and history depth for input and/or output ports. See Modifying Port Properties to learn how to assign a rate variable (data member) to a port. History depth can only be assigned to a *circular buffer* (users) type port. This is the first method called by the simulator before scheduling the simulation. All the parameter values are available to be read in the Setup() method and can be used to setup the rate if the rate depends upon a parameter value. The data members assigned to a port must not be used in this method. After running the model's Setup() method, the simulator allocates memories to all the ports and then schedule the simulation accordingly. The model itself must not allocate/reallocate/release memories for any of its data members assigned to a multi-rate port in this method. The Setup() method must return **true** on success and **false** on failure. A model can also post an error, warning, or an information message from inside the Setup() method. For an example, see the *Example Visual Studio Project for ModelBuilder* (users) shipped with SystemVue and used in *Quick Start* (users) section.

The Initialize() Method

Optionally the C++ model can override the **virtual bool Initialize()** method of its base class **AgilentEEsof::DFModel**, to perform any pre-simulation coding that model needs to perform. This method is called after the **virtual bool Setup()** and after simulator has calculated the schedule for the simulation run but before starting the simulation run. The model must not allocate/reallocate/release memories for any of its data members assigned to a multi-rate port in this method. The Initialize() method must return **true** on success and **false** on failure. A model can also post an error, warning, or an information message from inside the Initialize() method.

The Run() Method

The C++ model must override the **virtual bool Run()** method of its base class **AgilentEEsof::DFModel**, to perform any actions that model needs to perform during simulation. This method is called whenever simulator invokes the model based on the simulation schedule. The values of data members assigned to the inputs are already set by the simulator before calling the Run() and a C++ model can read the input values. The simulator reads the values of data members assigned to the outputs after calling Run() method.

> - For a multi-rate input use the index value '0' in [] operator to access the first data point received at that input port.
> - For a multi-rate output use the index value '0' in [] operator to to set the first output data point at the output port.
> - The maximum index value used in [] to access a data member assigned to a multi-rate port must be less than the corresponding rate of that port assigned in Setup() method.

The model must not allocate/reallocate/release memories for any of its data members assigned to a multi-rate port in this method. The Run() method must return **true** on success and **false** on failure. A model can also post an error, warning, or an information message from inside the Run() method. For an example, see the *Example Visual Studio Project for ModelBuilder* (users) shipped with SystemVue and used in *Quick Start* (users) section, some of the models in example project also contains multi-rate ports.

The Finalize() Method

Optionally the C++ model can override the **virtual bool Finalize()** method of its base class **AgilentEEsof::DFModel**, to perform any post-simulation coding that model needs to perform. This method is called by the simulator after the simulation run is completed. The model must not allocate/reallocate/release memories for any of its data members assigned to a multi-rate port in this method, the simulator takes care of releasing port and parameter memories by itself. The Finalize() method must return **true** on success and **false** on failure. A model can also post an error, warning, or an information message from inside the Finalize() method.

Posting Error, Warning or Information Messages

A model can post an error, warning, or information message using appropriate macro(s) below inside the Setup(), Initialize(), Run() or Finalize() methods or any non-static method of the model class which is called only inside Setup(), Initialize(), Run() or Finalize() methods.

- **POST_ERROR(<const_char_error>)**: The POST_ERROR macro will post a error to the error pane and simulation log. Additionally, the simulation will be forced to terminate. The input <const_char_error> to the macro must be of type **const char***.
- **POST_WARNING(<const_char_warning>)**: The POST_WARNING macro will post a warning to the error pane and simulation log. The input <const_char_warning> to the macro must be of type **const char***.
- **POST_INFO(<const_char_info>)**: The POST_INFO macro will post a informational message to the error pane and simulation log The input <const_char_info> to the macro must be of type **const char***.
- **POST_LOG(<const_char_info>)**: The POST_LOG macro will post to the simulation log. The input <const_char_info> to the macro must be of type **const char***.
- **POST_PROGRESS(<const_char_info>)**: The POST_PROGRESS macro will post a message to the status window, subsequent calls will replace the progress message. The input <const_char_info> to the macro must be of type **const char***.
- **CLEAR_PROGRESS()**: The CLEAR_PROGRESS macro will clear the messaged posted using the POST_PROGRESS macro.

The POST_ERROR, POST_WARNING, and POST_INFO macros pre-pend the message with the instance name of the model in SystemVue schematic which is posting the messages for easy debugging. It is advisable to not to post a warning and/or information message(s) for each invocation of Run() method, because Run() method is called several time depending upon the design being simulated. For a simple example use of POST_ERROR macro please read **Adder.cpp** file in the section *Adding a new Model to the Project* (users) in First Custom C++ Model Library section.

Reading or Writing Files

If you would like to read from or write to a file in your model then it is better to use file type parameters to specify the filenames. Please see Modifying Parameter Properties for details on how to create a File type parameter with browse button. However if you are not using file type parameter, then you must use absolute full path to the file location. For example use **c:\tmp\myfile.txt**, never use **myfile.txt**.

Using Inheritance

To inherit a model from an existing model, derive the class of new model from the existing model class with public access. There is no need to derive the new model from AgilentEEsof::DFModel because it will inherit DFModel from the existing parent model class. If the new model class needs to modify any of the Setup(), Initialize(), Run() or Finalize() methods then these methods must be declared virtual in the parent model class.

The derived model class must follow the following similar to the parent model class

- The derived model must use **DECLARE_MODEL_INTERFACE(<class name>);** in its header file, please see the section Writing Header file for the C++ Class.
- The derived model must define its interface inside DEFINE_MODEL_INTERFACE. The interface of the parent model could be added using the optional macro **ADD_PARENT_MODEL_INTERFACE(<ParentModelClass>)**, where <ParentModelClass> is the name of parent model class. If this macro is not used then, optionally, the derived model can add the complete interface itself. The use of **ADD_PARENT_MODEL_INTERFACE** is not required and it is provided to easily add the interface defined by the parent model.

The following header and cpp files show an example of a subtractor derived from Adder model shown in the section *Adding a new Model to the Project* (users) in First Custom C++ Model Library section.

Example Header File (subtractor.h) of a Derived Model

```
#pragma once
#include "Adder.h"
class Subtractor :
   public Adder
{
public:
   virtual bool Run();
   DECLARE_MODEL_INTERFACE(Subtractor);
};
```

Example cpp File (subtractor.cpp) of a Derived Model

```
#include "Subtractor.h"
DEFINE_MODEL_INTERFACE(Subtractor)
{
  ADD_PARENT_MODEL_INTERFACE(Adder);
  return true;
}
bool Subtractor::Run(void)
{
  Out = Gain * (In1 - In2);
  return true;
}
```

Writing Fixed Point Models

A model having at least one **AgilentEEsof::FixedPointCircularBuffer**, and/or **AgilentEEsof::FixedPointCircularBufferBus** input/output is considered as a fixed point model. A fixed point model class must be derived from **AgilentEEsof::DFModel** as well as an **interface** class **AgilentEEsof::DFFixedPointInterface** both with **public** access. The model class must also override the virtual **ERESULT SetOutputFixedPointParameters()** method to set output FixedPointParameters based on the model parameters or the FixedPointParameters of inputs. The FixedPointParameters for inputs are set by the simulator before calling **SetOutputFixedPointParameters** based on the output FixedPointParameters of the previous model in the design.

FixedPoint Inputs/Outputs

A model must use **AgilentEEsof::FixedPointCircularBuffer**, and/or **AgilentEEsof::FixedPointCircularBufferBus** to add a FixedPoint input or output, please see *SystemVue CircularBuffer Data Types* (users), and *SystemVue FixedPoint Data Type* (users) for further details. A data member of type AgilentEEsof::FixedPoint cannot be added as an input or an output.

Overriding SetOutputFixedPointParameters

The **SetParameters** method of AgilentEEsof::FixedPointCircularBuffer must be called for each single output and also for each output of a AgilentEEsof::FixedPointCircularBufferBus bus. You may also read the parameters of inputs in this method which are already set by the simulator. The **SetOutputFixedPointParameters** is called several times during simulation until convergence is achieved. In case of models whose output precision depends upon input precision, you may query that input has a valid FixedPointParameters or not using **AreParametersValid** method of the FixedPointCircularBuffer. An input may not have valid FixedPointParameters in first few iterations before convergence only when it is connected to a feedback loop. Even, if any of the input does not have valid FixedPointParameters you must set valid FixedPointParameters for all the outputs.

An example fixed point adder is shown below

```
// File AddFxp.h
#pragma once
#include "ModelBuilder.h"
#include "DFFixedPointInterface.h"
class AddFxp :
    public AgilentEEsof::DFModel, public AgilentEEsof::DFFixedPointInterface
{
public:
 /// Output Parameters
 int WordLength;
 int IntegerWordLength;
 AgilentEEsof::FixedPointEnums::Sign IsSigned;
 AgilentEEsof::FixedPointEnums::OverflowMode Overflow;
 AgilentEEsof::FixedPointEnums::QuantizationMode Quantization;
 int SaturationBits;
 /// input bus
 AgilentEEsof::FixedPointCircularBufferBus  dataIn;

 ///output
 AgilentEEsof::FixedPointCircularBuffer dataOut;
private:
 /// Accumulator for the sum
 /// AgilentEEsof::FixedPointValue is arbitray precision type. An object of
 /// FixedPointValue type may store a fixed-point value of arbitrary precision
 /// and binary point location without losing precision or magnitude (no quantization
 /// or overflow handling). This is suitbale for accumulating the sum. The
 /// overflow/quantization handling will be performed on dataOut[0] when we
 /// assign this accumulated sum to the dataOut[0]
 AgilentEEsof::FixedPointValue  m_fxpAccumulator ;
public:
 // This Macro is required for all classes derived from CDFModel
 DECLARE_MODEL_INTERFACE( AddFxp )
 //-------- Function Overloads --------
 bool    Run();    // Do the math
 bool    Initialize();

 ERESULT SetOutputFixedPointParameters();
};


// File AddFxp.cpp
#include "AddFxp.h"
DEFINE_MODEL_INTERFACE ( AddFxp )
{
 SET_MODEL_NAME( "AddFxp" );
 SET_MODEL_CATEGORY( "Math Scalar" );
 SET_MODEL_SYMBOL( "SYM_AddFxp" );
 ADD_MODEL_INPUT( dataIn );
 ADD_MODEL_OUTPUT( dataOut );
 WordLength = 16; // default value
 AgilentEEsof::DFParam cWL = ADD_MODEL_PARAMETER(WordLength);

 IntegerWordLength = 2; // default value
 AgilentEEsof::DFParam cIWL = ADD_MODEL_PARAMETER(IntegerWordLength);

 // Adding built in enumerations
 ADD_MODEL_PARAMETER(IsSigned);
 ADD_MODEL_PARAMETER(Quantization);
 ADD_MODEL_PARAMETER(Overflow);

 SaturationBits = 0; // default value
 AgilentEEsof::DFParam cSB = ADD_MODEL_PARAMETER(SaturationBits);

 return true;
}
ERESULT AddFxp::SetOutputFixedPointParameters()
{
 dataOut.SetParameters(WordLength,IntegerWordLength, IsSigned,Quantization,Overflow,SaturationBits
);
 return NOERROR_;
}
bool AddFxp::Initialize()
{
 if(WordLength <=0)
 POST_ERROR("Word Length must be greater than 0.");
 return true;
}
//--------------------------------------------------------------------------------
// Go
//  Here we do the math
//--------------------------------------------------------------------------------
bool AddFxp::Run()
{
 m_fxpAccumulator = 0;
 // accumulate the sum for all inputs on the bus without quantization/overflow
 // handling
 for(size_t szPort=0; szPort < dataIn.GetSize(); szPort++)
 {
 m_fxpAccumulator += dataIn[szPort][0];
 }
 // assign the accumulated sum to output, this will cause quantization/overflow handling
 dataOut[0] = m_fxpAccumulator;
 return true;
}
```

Writing a Fixed Point Model for Fixed Point Analysis

When *Data Flow Analysis options* (sim) are set to collect fixed point analysis data, SystemVue collects and analyze data at the output ports to collect the information needed for *fixed point analysis table* (sim) at the end of each execution of **Run** method. This means, to detect an overflow and underflow at the output, the overflow and quantization flags for the fixed point data at each output are set properly at the end of **Run** method. For example, if we have written the Run method in the AddFxp example as follows then the overflow and quantization flags may not be set properly

```
/// This code is not a recommended practice and may result in incorrect information
```

```
/// in the Fixed Point Analysis Table.
bool AddFxp::Run()
{
 dataOut[0] = 0;
 // Accumulate the sum for all inputs directly in the dataOut[0];
 // this may cause incorrect overflow/underflow information in
 // fixed point analysis table
 for(size_t szPort=0; szPort < dataIn.GetSize(); szPort++)
 {
 dataOut[0] += dataIn[szPort][0];
 }
 return true;
}
```

The reason is that if the input bus for the adder has more than one inputs, then there may be quantization or overflow during the accumulation phase (for loop) except in the last assignment to dataOut[0]. Since the last assignment results in no overflow/quantization therefore at the end of **Run** method these flags are not set at the output properly. To avoid this problem make sure to keep one assignment per output port data point in the Run method for models which can cause overflow and/or quantization. There are models which will generally do not cause an underflow or quantization (such as AND, OR, XOR), this restriction can be relaxed for such models. In the original AddFxp code above, an object of *Fixed Point Value* (users) is used as an accumulator and the final result is assigned to dataOut[0] at the end resulting in proper overflow and/or quantization flag values at the end of Run() method.

For more example fixed point models, see the *Example Visual Studio Project for ModelBuilder* (users) shipped with SystemVue and used in *Quick Start* (users) section.

Writing Timed Data Flow Models

SystemVue supports two domains of models, numeric (*untimed* (sim)) models and *timed* (sim) models, for representing different timing behavior. The above sections in this document mainly focus on numeric (untimed) models. In this section, we discuss how to write timed C++ models. For introduction to timed data flow models, we refer the users to *Timing Method* (sim).

Timed Data Flow Model Class

A timed model class must derive from **AgilentEEsof::TimedDFModel**, which is the base class for timed C++ model. AgilentEEsof::TimedDFModel is defined in \ModelBuilder\include\SystemVue\TimedDFModel.h in the SystemVue installation directory. AgilentEEsof::TimedDFModel inherits from AgilentEEsof::DFModel to provide additional firing count property for timing calculation. The firing count (**TimedDFModel::m_iFiringCount**) records the number of executions (runs) of the model during the simulation, and it is initialized to 0. The member methods of AgilentEEsof::TimedDFModel are described as follows.

- **void Advance()**: Increase firing count after each execution. When using the timed model in SystemVue, TimedDFModel::Advance() is automatically called after each execution, i.e., called after each overrided DFModel::Run(). When using the timed model outside SystemVue, users have to manually invoke TimedDFModel::Advance().
- **unsigned long long GetCount()**: Query the current firing count.
- **ERESULT CalculateLatency()**: See Overriding Latency Calculation.
- **ERESULT PropagateCharacterizationFrequency()**: See Overriding Characterization Frequency Propagation

Timed Circular Buffer

SystemVue provides ***AgilentEEsof::TimedCircularBuffer<T>*** (users) for a timed model to access *time stamps* (sim) of the input (or output) data samples and to set sample rate and latency information. In general, data flow production and consumption rates as well as sample rates (for particular TimedCircularBuffer) should be set in TimedDFModel::Setup(). To set sample rate (equivalently, 1 / time step), use **SetSampleRate** or **SetTimeStep** method of *AgilentEEsof::TimedCircularBuffer<T>* (users). To set latency of the timed model, use **SetStartTime** method of *AgilentEEsof::TimedCircularBuffer<T>* (users) in TimedDFModel::CalculateLatency(). In TimedDFModel::Initialize() and TimedDFModel::Run(), use **GetSampleRate**, **GetTimeStep**, and **GetStartTime** methods of *AgilentEEsof::TimedCircularBuffer<T>* (users) to get the sample rate, time step, and start time associated with the timed circular buffer. In TimedDFModel::Run(), use **GetTime** method of *AgilentEEsof::TimedCircularBuffer<T>* (users) to get the time stamp of a particular sample on the timed circular buffer.

The following SineGenerator example shows how to write a simple sine generator model that sets simulation sample rate based on the *SampleRate* parameter and generates timed sine wave based on *Amplitude*, *Frequency*, and *Phase* parameters.

```
//SineGenerator.h
#pragma once
#include "ModelBuilder.h"
#include "SystemVue/TimedDFModel.h"
#include "SystemVue/TimedCircularBuffer.h"
class SineGenerator : public AgilentEEsof::TimedDFModel //derive from AgilentEEsof::TimedDFModel
{
public:
 DECLARE_MODEL_INTERFACE( SineGenerator )
 virtual bool Run();
 virtual bool Setup();
       //parameters
 double Amplitude;
 double Frequency;
 double Phase;
 double SampleRate;
       //declare output as a timed circular buffer of type double
 AgilentEEsof::TimedCircularBuffer<double> output;
};
```

```
//SineGenerator.cpp
#include "SineGenerator.h"
#define TWOPI 6.283185307179586476925286766559005768394338798750211
#ifndef SV_CODE_GEN
DEFINE_MODEL_INTERFACE( SineGenerator )
{
       //Add TimedCircularBuffer output as a model output
 ADD_MODEL_OUTPUT( output );
 AgilentEEsof::DFParam paramAmp = ADD_MODEL_PARAM( Amplitude );
 paramAmp.SetDefaultValue( "1.0" );
 AgilentEEsof::DFParam paramFreq = ADD_MODEL_PARAM( Frequency );
 paramFreq.SetDefaultValue( "5e3" );
 AgilentEEsof::DFParam paramPhase = ADD_MODEL_PARAM( Phase );
 paramPhase.SetDefaultValue( "0" );
 AgilentEEsof::DFParam paramSR = ADD_MODEL_PARAM( SampleRate );
 paramSR.SetDefaultValue( "1e6" );
 return true;
}
#endif
bool SineGenerator::Setup()
{
 bool bStatus = true;
 if ( SampleRate > 0 )
 {
               //Use TimedCircularBuffer::SetSampleRate method to set the output sample rate in
Setup()
 output.SetSampleRate( SampleRate );
 }
 else
 {
 POST_ERROR( "SampleRate must be greater than 0." );
 bStatus = false;
 }
 return bStatus;
}
bool SineGenerator::Run()
{
 bool bStatus = true;

       //Use TimedCircularBuffer::GetTime method to get the time stamp of the output sample
```

```
        //In output.GetTime( 0, m_iFiringCount ), 0 means the 0th output sample of each firing
(run), and TimedDFModel::GetCount returns the current firing count.
 output[0] = Amplitude * sin( TWOPI * Frequency * output.GetTime( 0, GetCount() ) + Phase );

        return bStatus;
}
```

Overriding Latency Calculation

The derived timed model class can override the virtual **ERESULT CalculateLatency()** method to set the start time of output **TimedCircularBuffer** based on the start time and time step of input **TimedCircularBuffer** and model parameters. If the derived timed model does not override this method, the start time of the output is default to the start time of the input.

The following TimedDownSampler shows an example that overrides TimedDFModel::CalculateLatency(). The input samples are downsampled by *Factor*. For each firing (run), only the *Phase* th sample among *Factor* input samples is sent to the output. As a result, to make the behavior causal, the time stamp of the first output sample should be delayed by *Phase* * input time step for causality.

```
//TimedDownSampler.h
#pragma once
#include "ModelBuilder.h"
#include "SystemVue\TimedDFModel.h"
#include "SystemVue\TImedCircularBuffer.h"
class TimedDownSampler : public AgilentEEsof::TimedDFModel
{
public:
 DECLARE_MODEL_INTERFACE( TimedDownSampler )
 virtual bool Run();
 virtual bool Setup();
        //Override default latency calculation
 ERESULT CalculateLatency();
 int Factor;
 int Phase;
 AgilentEEsof::TimedCircularBuffer<double> input;
 AgilentEEsof::TimedCircularBuffer<double> output;
};
```

```
//TimedDownSampler.cpp
#include "TimedDownSampler.h"
DEFINE_MODEL_INTERFACE( TimedDownSampler )
{
 AgilentEEsof::DFParam paramFactor = ADD_MODEL_PARAM( Factor );
 paramFactor.SetDefaultValue( "2" );
 AgilentEEsof::DFParam paramPhase = ADD_MODEL_PARAM( Phase );
 paramPhase.SetDefaultValue( "0" );
 ADD_MODEL_INPUT( input );
 ADD_MODEL_OUTPUT( output );
 return true;
}
bool TimedDownSampler::Setup()
{
        bool bStatus = true;
 if ( Factor < 1 )
        {
 POST_ERROR("Phase should be greater than 1.");
                bStatus = false;
        }
        //Set input data flow rate to Factor
        input.SetRate( (size_t)Factor );
 if( Phase >= Factor || Phase < 0 )
        {
 POST_ERROR("Phase should be greater than or equal to 0 and less than Factor");
                bStatus = false;
        }

        return bStatus;
}
ERESULT TimedDownSampler::CalculateLatency()
{
        //For causality, set output start time to be the input start time + Phase * input time step
 output.SetStartTime( input.GetStartTime ()+ input.GetTimeStep() * Phase );
 return NOERROR_;
}
bool TimedDownSampler::Run()
{
 output[0] = input[ (size_t)Phase ];
 return true;
}
```

Using Envelope Signal in Timed Data Flow Model

SystemVue provides *AgilenEEsof::EnvelopeSignal* (users) data type and corresponding *AgilentEEsof::EnvelopeCircularBuffer* (users) for writing models which requires complex envelope signals.

The following EnvelopeToReal example shows how to convert an envelope signal (which can represent either a real signal or a complex envelope signal) to real signal.

```
//EnvelopeToReal.h
#pragma once
#include "ModelBuilder.h"
#include "SystemVue\TimedDFModel.h"
#include "SystemVue\EnvelopeSignal.h"
class EnvelopeToReal : public AgilentEEsof::TimedDFModel
{
 DECLARE_MODEL_INTERFACE( EnvelopeToReal )
 virtual bool Run();
 //Envelope signal
 AgilentEEsof::EnvelopeCircularBuffer input;
 //Real signal
 AgilentEEsof::CircularBuffer<double> output;
};
```

```
//EnvelopeToReal.cpp
#include "EnvelopeToReal.h"
DEFINE_MODEL_INTERFACE( EnvelopeToReal )
{
 ADD_MODEL_INPUT( input );
 ADD_MODEL_OUTPUT( output );
 return true;
}
bool EnvelopeToReal::Run()
{
 //If input represents a real signal (based on whether the characterization frequency is equal to
0)
 if ( input.GetCharacterizationFrequency() == 0 )
 {
 //Use EnvelopeSignal::real() to get the value of the real signal
 output[0] = input[0].real();
 }
 //Otherwise, input represents a complex envelope with associated characterization frequency
 else
 {
 //Use EnvelopeSignal::ConvertToReal to convert the complex envelope to real signal
 output[0] = input[0].ConvertToReal( input.GetCharacterizationFrequency(), input.GetTime(0,
GetCount()) );
 }
 return true;
}
```

Overriding Characterization Frequency Propagation

The derived timed model class can override the virtual **ERESULT PropagateCharacterizationFrequency()** method to set the characterization frequency of output EnvelopeCircularBuffer based on the characterization frequency of input EnvelopeCircularBuffer and model parameters. If the derived model does not override this method, the characterization frequency of output EnvelopeCircularBuffer is default to the maximum characterization frequency among input EnvelopeCircularBuffers.

The following Modulator example up converts baseband I-Q complex sample to complex envelope signal at *CarrierFrequency*, and use TimedDFModel::PropagateCharacterizationFrequency() to set the output characterization frequency.

```
//Modulator.h
#pragma once
#include "ModelBuilder.h"
#include "SystemVue\TimedDFModel.h"
#include "SystemVue\EnvelopeSignal.h"
class Modulator : public AgilentEEsof::TimedDFModel
{
 DECLARE_MODEL_INTERFACE( Modulator )
 virtual bool Run();
 ERESULT PropagateCharacterizationFrequency();
 double CarrierFrequency;
 //Complex baseband I-Q signal
 AgilentEEsof::DComplexCircularBuffer input;

 //Envelope signal
 AgilentEEsof::EnvelopeCircularBuffer output;
};
```

```
//Modulator.cpp
#include "Modulator.h"
DEFINE_MODEL_INTERFACE( Modulator )
{
 AgilentEEsof::DFParam paramCarrierFrequency = ADD_MODEL_PARAM( CarrierFrequency );
 paramCarrierFrequency.SetDefaultValue( "1e6" );
 ADD_MODEL_INPUT( input );
 ADD_MODEL_OUTPUT( output );
 return true;
}
ERESULT Modulator::PropagateCharacterizationFrequency()
{
 //Set output envelope signal characterization frequency to be carrier freqnecy
 output.SetCharacterizationFrequency( CarrierFrequency );
 return NOERROR_;
}
bool Modulator::Run()
{
 //Assign input complex baseband I-Q value to output envelope signal with associated carrier
frequency
 output[0] = input[0];
 return true;
}
```

Controlling Simulation

For more details, see *Simulation Control* (sim).

SystemVue provides **DFSinkControl** in \ModelBuilder\include\SystemVue\SimulationControl.h in the SystemVue installation directory.

The member methods of AgilentEEsof::DFSinkControl are described as follows.

- **bool Initialize(DFModel* pModel, unsigned long long iStartSample, unsigned long long iStopSample)** : Initialize DFSinkControl for untimed sink. *pModel* is the pointer to the DFModel that owns the DFSinkControl.
- **bool Initialize(DFModel* pModel, double dStartTime, double dStopTime, double dTimeStep, double dFirstTimeStamp)** : Initialize DFSinkControl for untimed sink. *pModel* is the pointer to the DFModel that owns the DFSinkControl.
- **bool CollectData()** : Surround data collection in DFModel::Run() with the following code to ensure proper data collection as specified in the Initialize method. Data collection should be performed if CollectData() returns true, otherwise, data collection should be avoided.

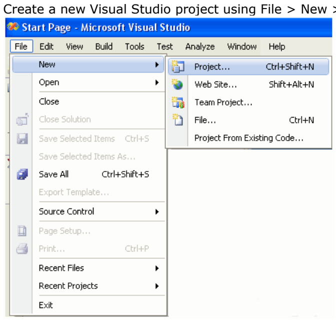
```
if ( sink_control_object.CollectData() )
{
       //data collection code ...
}
```

Building Your First Custom C++ Model Library

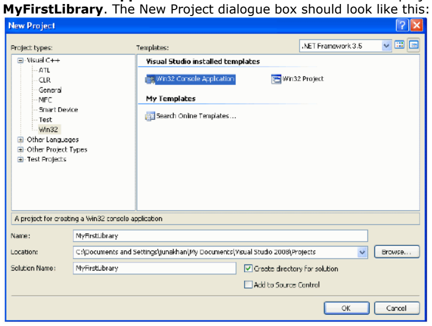
In this section, we will build a simple adder with two inputs of type "double". The model will add the two inputs and then multiply the result with a "Gain" parameter before passing it to the output. It is assumed that you have already installed SystemVue and the Microsoft Visual Studio version mentioned in the ***Requirements*** (users) section. The following section also assumes that you have installed SystemVue at **C:\Program Files\SystemVue2009.08**.

Setting Up a New Visual Studio Project

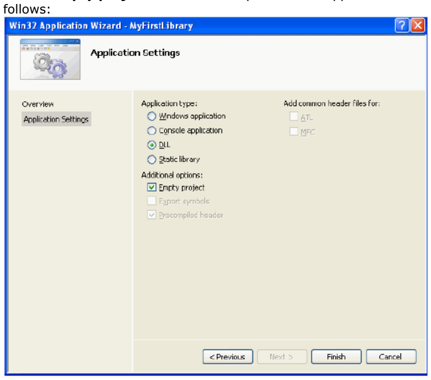
1. Start Microsoft Visual Studio.
2. Create a new Visual Studio project using File > New > Project as shown below.



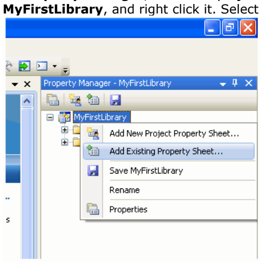
3. In the New Project dialogue box choose **Win32** under Visual C++ and then choose **Win32 Console Application**. In this case we will call our first project **MyFirstLibrary**. The New Project dialogue box should look like this:



4. Click **Ok** and in the next dialogue select **Next >**
5. In the dialogue under **Application Settings** choose **DLL** as the Application type and choose **Empty project** as Additional Options. Your application settings should be as follows:



6. Click **Finish**.
7. Click **View > Property Manager**, this should open Visual Studio **Property Manager**.
8. In **Property Manager**, select the project you want to setup, in our case it is **MyFirstLibrary**, and right click it. Select **Add Existing Property Sheet**:



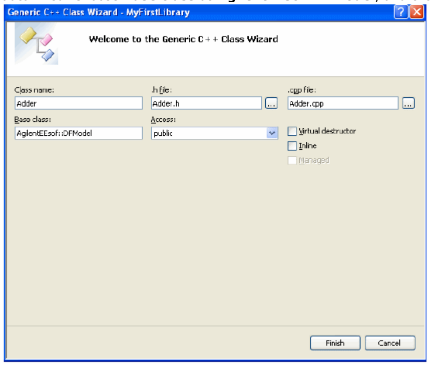
9. Browse to your SystemVue installation, and under the **ModelBuilder** directory, choose **Model Builder.vsprops** and open it. With a default installation this should be located at **C:\Program Files\SystemVue2009.08\ModelBuilder**. We are now done with Property Manager.

> ⚠ Note, The property sheet assigned to a project must be updated whenever you choose to update SystemVue, especially if the SystemVue installation location is updated.

Adding a new Model to the Project

1. Right click on the project **MyFirstLibrary** and select **Add > Class**. In the Add Class dialogue box select **C++** as the category and **C++ Class** as the template. Click **Add**.
2. In the **Generic C++ Class Wizard** dialogue box, add the **Class name** of your model, in our example we will call it **Adder**, the **.h file** and **.cpp file** fields will be

auto-filled. Choose **Base class** as **AgilentEEsof::DFModel**, and click **Finish**.



⚠ All SystemVue Model classes must be derived from **AgilentEEsof::DFModel** with **public** access

3. Modify the added **Adder.h** file so that it looks like:

```
#pragma once
#include "ModelBuilder.h"
class Adder :
 public AgilentEEsof::DFModel
{
public:
 double In1, In2; // Inputs
 double Out; // Outputs
 double Gain; // Parameters
 DECLARE_MODEL_INTERFACE(Adder); // Declare modelbuilder interface
 virtual bool Run();   // Method for scientific code during each invocation of the model
 virtual bool Initialize(); // This method is invoked before the start of simulation
};
```

- You must include **ModelBuilder.h**.
- You must add macro **DECLARE_MODEL_INTERFACE(<ClassName>);** with public access.
- The data members for parameters and input/outputs must have public access.

Modify the the Adder.cpp file so that it looks like:

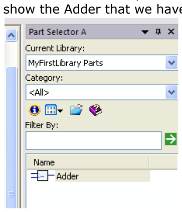
```
#include "Adder.h"
DEFINE_MODEL_INTERFACE(Adder)
{
 ADD_MODEL_INPUT(In1);
 ADD_MODEL_INPUT(In2);
 ADD_MODEL_OUTPUT(Out);
 Gain = 0; // Default Value
 ADD_MODEL_PARAM(Gain);
 return true;
}
bool Adder::Initialize()
{
 if (Gain ==0)
 {
 POST_ERROR("The value of Gain cannot be == 0");
 return false;
 }
 return true;
}
bool Adder::Run()
{
 Out = Gain * (In1 + In2);
 return true;
}
```

- Use the **ADD_MODEL_INPUT(<data member>);** macro to add a data member as input.
- Use the **ADD_MODEL_OUTPUT(<data member>);** macro to add a data member as output.
- Use the **ADD_MODEL_PARAM(<data member>);** macro to add a data member as a parameter.
- Inputs, outputs and parameters can only be added inside **DEFINE_MODEL_INTERFACE(<class name>)** macro.
- Use **POST_ERROR** macro to post an error.

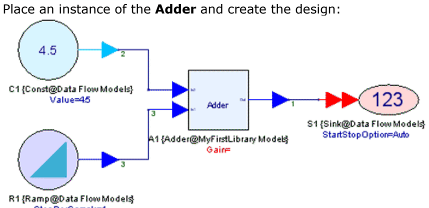
Build the solution, using either the Debug or Release configuration by right clicking on the solution and selecting **Build Solution**. A successful build should create **<project name>.dll** in the Debug and Release directories respectively, in our case it will be **MyFirstLibrary.dll**.

Using the Model in SystemVue

1. Start SystemVue using a Blank template
2. Click **Tools > Library Manager...**.
3. In the **Library Manager** dialogue box, select **Add From File**.
4. Browse to your Project location and then into either the **Debug** or **Release** sub directory (use the configuration that you chose to Build the project).
5. Change the **File of Type** to "SystemVue DLL Libraries (.dll)" and select the **<project name>.dll**. In our case it will be **MyFirstLibrary.dll**.
6. Click **Open**. Scroll down to see that the library has been added and is shown in the list.
7. Click **Close**.
8. Under **Part Selector**, in **Current Library** choose **MyFirstLibrary Parts**, this will show the Adder that we have created.



9. Place an instance of the **Adder** and create the design:

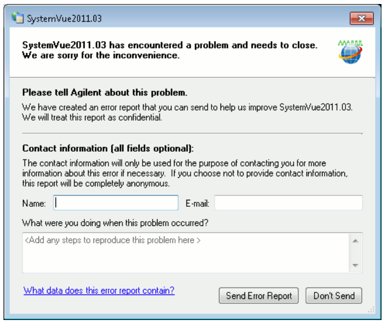


10. Simulate the design. The design will give an expected error about the value of Gain == 0. This is the error we have posted in our **Initialize()** method in **Adder.cpp** file above.
11. Change the value of Gain to a non-zero value and successfully simulate the design.

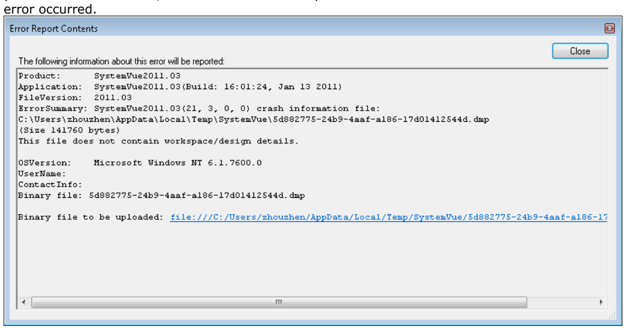
⚠ SystemVue uses the DLL library name to name the Part, Model and Enum libraries. The model builder DLLs that you load must have unique names. Please read *Defining the Model Library Properties* (users) to override this default behavior.

What to Do if the Model Terminates SystemVue Unexpectedly

When SystemVue terminates unexpectedly during simulation, the following error report dialog window will pop up, and you can do one of two things:



- Submit the error report
- If you use Visual Studio IDE
  - open the following **Error Report Contents** dialog window by **clicking the " What data does this error report contain?" link**.
  - click the link to the **.dmp** file (Visual Studio should come up and load the **.dmp**)
  - start debugging by selecting "Start Debugging" under the Debug menu or by pressing the F5 key. If the problem that caused the unexpected termination is in your custom model code, Visual Studio should stop at the line of code where the error occurred.



Loading and Debugging a C++ Model Library

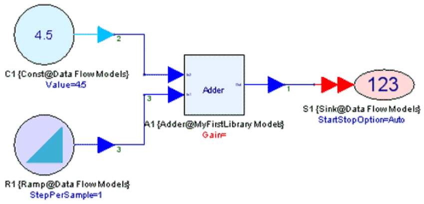
Loading a C++ Model Library

To load your model builder DLL, use the **Library Manager**. See *Using the Library Manager* (users) documentation for more details.

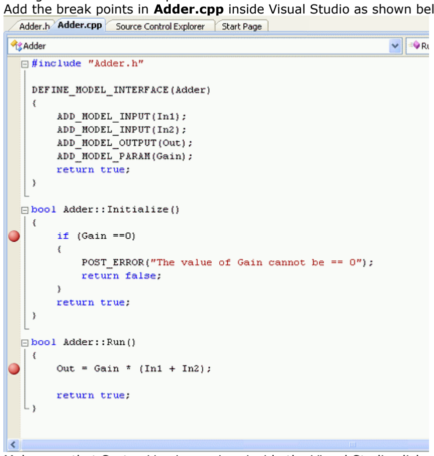
Debugging Data Flow C++ Models

Debugging a Data Flow C++ model requires the corresponding library to be built with **Debug** solution configuration in Visual Studio. For hands on learning we will be using the Adder model that we have developed in the section *Adding a new Model to the Project* (users) above.

1. Build the library with **Debug** configuration, load the library in SystemVue and create the design using the Adder model as shown in figure below

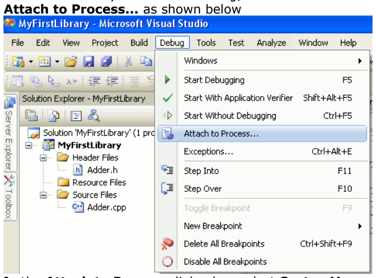


2. Change the value of Gain parameter to a non-zero value and save the design.
3. Add the break points in **Adder.cpp** inside Visual Studio as shown below

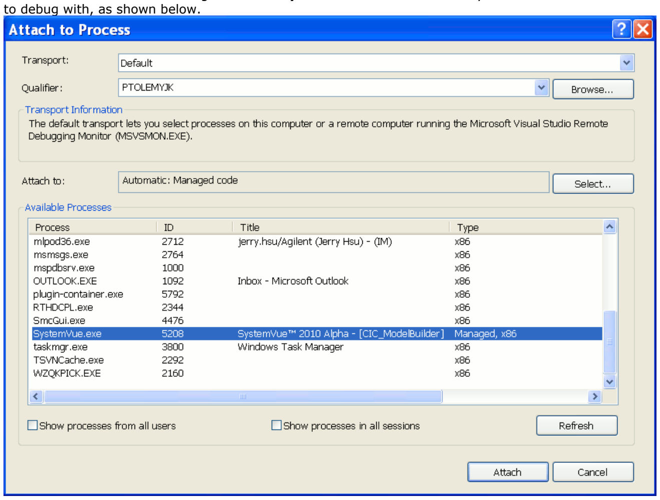


```
#include "Adder.h"

DEFINE_MODEL_INTERFACE(Adder)
{
    ADD_MODEL_INPUT(In1);
    ADD_MODEL_INPUT(In2);
    ADD_MODEL_OUTPUT(Out);
    ADD_MODEL_PARAM(Gain);
    return true;
}

bool Adder::Initialize()
{
    if (Gain ==0)
    {
        POST_ERROR("The value of Gain cannot be == 0");
        return false;
    }
    return true;
}

bool Adder::Run()
{
    Out = Gain * (In1 + In2);

    return true;
}
```

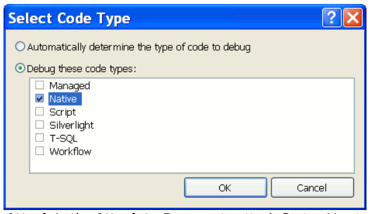
4. Make sure that SystemVue is running, inside the Visual Studio click on **Debug -> Attach to Process...** as shown below



5. In the **Attach to Process** dialog box select **SystemVue.exe** instance that you want to debug with, as shown below.



1. Click **Select...** to bring up **Select Code Type** dialog box.
2. Select **Debug these code types:** and then select **Native**. Click **OK** button to save settings and close **Select Code Type** dialog box.



6. Click **Attach** in the **Attach to Process** to attach SystemVue to the Visual Studio debugger.
7. In the SystemVue instance, that is now attached to Visual Studio, start the simulation. This will invoke the break point in Visual Studio inside **Adder::Initialize()** function. Hit continue, and next break point will be in **Run()** method, hit continue again, this will again stop inside **Run()** but for the next input data point. Keep debugging in Visual Studio as you do for any other C++ code. Read visual studio documentation to learn more about how Visual Studio debugger works.
8. Remove break point from inside **Run()** method and hit continue again, this will finish the SystemVue simulation.

Making Changes in C++ Model while SystemVue is Running

If you make any change in your code in Visual Studio and try to build the project while the corresponding DLL is still loaded, then Visual Studio build process will fail complaining that it cannot open the corresponding DLL. To build the Visual Studio project without closing SystemVue, *remove (unload)* (users) the corresponding DLL from SystemVue using the **Library Manager**. You may need to *Add (load)* (users) the DLL again in SystemVue after re-building the DLL. See *Using the Library Manager* (users) documentation for more details. Optionally you could close SystemVue, build the project and then restart SystemVue without having the need to unload/load the DLL library.

Quick start

This quick start section will cover building an example Visual Studio project shipped with SystemVue, loading the newly built dll in SystemVue and running the simulation using example workspaces. The later sections will cover:

- *setting up a new Visual Studio project* (users) to build custom C++ models
- *writing C++ models* (users)
- *debugging C++ models* (users)

SystemVue is shipped with an example Visual Studio project in the directory **C:\Program Files\SystemVue2011.03\ModelBuilder\SystemVue Model Builder**, where C:\Program Files\SystemVue2009.08 is the directory where SystemVue is installed. This project contains source code for several C++ models. Some of those model are shown in the following table.

Model	Description	Example Workspace using the Model
AddCx	Two input complex adder	Examples\Model Building\C Modeling\Simple Model Builder Example.wsv
FIR	Floating point FIR filter, functionally equivalent to *FIR model* (algorithm)	Examples\Model Building\C Modeling\Simple Model Builder Example.wsv
CIC_Filter	Floating point cascaded integrator-comb (CIC) filter	Examples\Model Building\CIC Filter.wsv
UpSample	Floating point upsampler, similar to *UpSample model* (algorithm)	Examples\Model Building\C Modeling\Simple Model Builder Example.wsv

Warning
Before opening the examples in the above table, you must compile the example Visual Studio project and load the generated **Custom.dll** file in SystemVue

Compiling the Example Visual Studio Project

1. Copy the **C:\Program Files\SystemVue2009.08\ModelBuilder\SystemVue Model Builder** directory to any location on the same computer where SystemVue is installed. A good location to copy this project can be the default Visual Studio projects directory such as **My Documents\Visual Studio 2008\Projects** for Visual Studio 2008.
2. In Visual Studio, open the solution file **SystemVue Model Builder.sln** located in the directory you just copied. This solution contains a Visual Studio project named **Custom**.
3. Observe the code in the **Custom** project. You may also look at the ReadMe.txt.
4. Build the library by clicking **Build -> Build Solution**, by default the **Debug** configuration will be built. A successful build of this project will create a library named **Custom.dll** under **SystemVue Model Builder\Debug** directory.

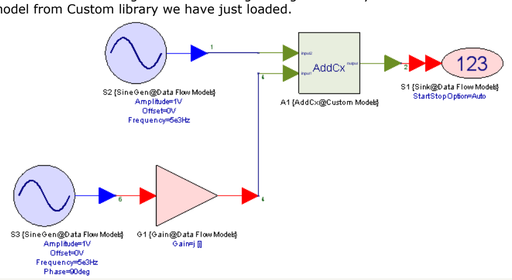
Loading the Custom Library into SystemVue

Next, you'll need to load the custom library into SystemVue. To do this, follow the steps below:

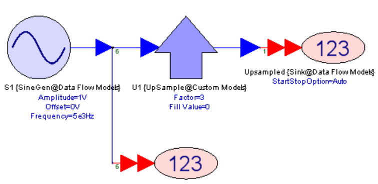
1. Start SystemVue with **Blank** Workspace.
2. Click **Tools > Library Manager...**.
3. In the **Library Manager** dialogue box, select **Add From File**.
4. Browse to **SystemVue Model Builder\Debug** directory.
5. Change the **File of Type** to "SystemVue DLL Libraries (.dll)" and select **Custom.dll**.
6. In the SystemVue Part Selector, you will now see a new library named **Custom Parts**.

Simulating the Example WorkSpace

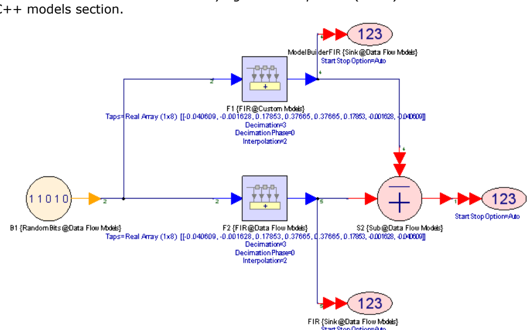
1. Open **Examples\Model Building\C Modeling\Simple Model Builder Example.wsv** in SystemVue.
2. The workspace contains three designs
   1. **AddCx Test** containing instance A1 using auto-generated symbol and AddCx model from Custom library we have just loaded.



   2. **UpSample Test** containing instance U1. In this case, UpSample model from Custom library was added to SystemVue built in part UpSample using *Manage Model* (users) option.



   3. **FIR Test** containing instance F1. In this case, the pre-existing FIR symbol was hard-coded in the C++ model. Assigning the existing symbol to a model in C++ code is covered in the section *Modifying Model Properties* (users) in Data Flow C++ models section.



3. Simulate each design, and observe the results.

Requirements

- SystemVue must be installed on the machine where you will be building the Custom C++ model library.
- The SystemVue C++ Model Builder requires either:
  - **Microsoft Visual C++ 2008 Express Edition** (freely available from Microsoft)
  - **Microsoft Visual Studio C++ 2008 with SP1**

Supported Data Types

There are certain restrictions on using data types for inputs, outputs, and/or parameters. You are free to use any valid C++ data type object if it is not used as an input, output, or a parameter.

Data Types Used as Parameters

A C++ model can support only the following types for class data members that will be used as parameters. These class data members must be declared with **public** access in the class declaration:

- **C++ Built In Scalar Data Types:** The C++ data types int, double, float, std::complex<double>, std::complex<float>, bool, and char * are supported as scalar parameters. Note that char * is used as scalar parameter to represent a character string or file name type parameter.
- **C++ Built In Pointer Data Types:** The C++ pointer data types int*, double*, and std::complex<double>* are supported as array parameters. Each pointer data type must be accompanied with an **unsigned** type data member to hold the size of array set by the simulator.
- **SystemVue Matrix Data Type:** The Matrix is supported for int, float, double, std::complex<float> and std::complex<double> version of AgilentEEsof::Matrix.
- **SystemVue Built In Enumerations:**
  - **AgilentEEsof::QueryEnum**: Possible values are AgilentEEsof::QUERY_NO and AgilentEEsof::QUERY_YES.
  - **AgilentEEsof::BooleanEnum**: Possible values are AgilentEEsof::BOOLEAN_FALSE and AgilentEEsof::BOOLEAN_TRUE.
  - **AgilentEEsof::SwitchEnum**: Possible values are AgilentEEsof::SWITCH_OFF and AgilentEEsof::SWITCH_OFF.
  - **AgilentEEsof::FixedPointEnums::Sign**: Possible values are AgilentEEsof::FixedPointEnums::UNSIGNED and AgilentEEsof::FixedPointEnums::TWOS_COMPLEMENT.
  - **AgilentEEsof::FixedPointEnums::QuantizationMode**: Possible values are AgilentEEsof::FixedPointEnums::ROUND AgilentEEsof::FixedPointEnums::ROUND_ZERO, AgilentEEsof::FixedPointEnums::ROUND_MINUS_INFINITY, AgilentEEsof::FixedPointEnums::ROUND_INFINITY, AgilentEEsof::FixedPointEnums::ROUND_CONVERGENT, AgilentEEsof::FixedPointEnums::TRUNCATE, and AgilentEEsof::FixedPointEnums::TRUNCATE_ZERO.
  - **AgilentEEsof::FixedPointEnums::OverflowMode**: Possible values are AgilentEEsof::FixedPointEnums::SATURATE, AgilentEEsof::FixedPointEnums::SATURATE_ZERO, AgilentEEsof::FixedPointEnums::SATURATE_SYMMETRICAL, AgilentEEsof::FixedPointEnums::WRAP, and AgilentEEsof::FixedPointEnums::WRAP_SIGN_MAGNITUDE.

    > **Note**
    > At your option, to avoid long nested namspace such as "AgilentEEsof::FixedPointEnums" you may use **using** directive in cpp file. The use of **using** directive is not encouraged in .h files.
- **User Defined Enumerations:** A user defined C++ enum can also be used as a parameter, the enumeration type needs to be specified when adding such an enumeration for proper type conversions. The details will be given in later sections.

Data Types Used as Inputs/Outputs

A C++ model can support only the following types for class data members that will be used as inputs/outputs. These class data members must be declared with **public** access in the class declaration:

- **C++ Built In Data Types as Uni-rate Inputs/Outputs:** The C++ data types int, double, and std::complex<double> are supported as uni-rate inputs/outputs.
- **C++ Built In Pointer Data Types as Multi-rate Inputs/Outputs:** The C++ pointer data types int*, double*, and std::complex<double>* are supported as multi-rate inputs/outputs. For these data types an **unsigned** type rate variable may be added to specify the rate; the default rate for each input/output using these data types is "1".
- **SystemVue CircularBuffer Data Types:** SystemVue supports highly effecient built in CircularBuffer data types to implement inputs/outputs for better performance and ease of coding. It is highly recommended to use the CircularBuffer data types as inputs/outputs instead of C++ built in data types. The CircularBuffer data types are multi-rate in nature; to implement a uni-rate model use rate=1.
- **SystemVue CircularBufferBus Data Types:** A bus of CircularBuffer inputs/outputs. The CircularBufferBus data types are the only way to implement a bus input or bus output.
- **SystemVue TimedCircularBuffer<T> Data Types:** The templated TimedCircularBuffer<T> data types are similar to CircullarBffer data types but they are also able to access time stamps, and to set/get sample rate. TimedCircularBuffer should only be use inside *TimedDFModel* (users).
- **SystemVue EnvelopeCircularBuffer Data Types:** Inherits from **TimedCircularBuffer< EnvelopeSignal >** and uses a private member **double m_dFc** to store the characterization frequency associated with the envelope signal. Please read Envelope Signal Type for more details.

CircularBuffer Data Types

A templated CircularBuffer< T > data type is multi-rate by design, hence it is considered as collection of individual data points, which can be referenced using the [] operator. The index 0 for the [] operator points to the oldest sample. A CircularBuffer< T > data type can be used exactly the same way as an array of the same basic data type. For example, AgilentEEsof::CircularBuffer< double > can be used exactly the same way as a double *. By default, the rate of a CircularBuffer is set to **1**. To change the rate, use the **void SetRate(size_t iRate)** method. The rate value can be queried using the **size_t GetRate() const** accessor method. SystemVue handles all the memory allocation/deallocation for the CircularBuffer< T > data types. This memory is guaranteed to be allocated before the first invokation of the *Run()* (users) method of a model. Therefore, the [] operator used to access the individual data values inside the CircularBuffer< T > must only be called inside the *Run()* (users) method of your model. Using the [] operator outside the *Run()* (users) method would cause access to NULL memory location, which will result in a crash. On the other hand, the **SetRate** method must be called inside *Setup()* (users) method if you want to set the rate of the CircularBuffer to a value other than the default one (1) and the **GetRate** method can be called anywhere in your code. Inside the *Run()* (users) method of your model, you can also query the CircularBuffer< T > to find out whether the corresponding input/output is connected. This is done using the **IsConnected()** method and is useful for *inputs/outputs that are set to be optional* (users). Sometimes, there is a need to access input samples older than what you can get based on input's multi-rate properties. In this case, the method **void SetHistoryDepth(size_t iHistoryDepth)** can be used (only inside *Setup()* (users) method of a model) to set the number of samples that need to be stored in the CircularBuffer< T > including the most recent sample (if this method is not used the number of samples stored in the CircularBuffer<T> is equal to its rate). Index 0 for the [] operator will point to the oldest sample in the history. The argument **iHistoryDepth** must be greater than or equal to the CircularBuffer<T> rate. The method **size_t GetHistoryDepth()** can be used to access the history depth.

The following CircularBuffer<T> data types are predefined using typedefs and are only supported types for use as inputs/outputs:

- **AgilentEEsof::BoolCircularBuffer**: Stores bool objects (behaves like bool *). Same as AgilentEEsof::CircularBuffer< bool >.
- **AgilentEEsof::IntCircularBuffer**: Stores int objects (behaves like int *). Same as AgilentEEsof::CircularBuffer< int >.
- **AgilentEEsof::DoubleCircularBuffer**: Stores double objects (behaves like double

*). Same as AgilentEEsof::CircularBuffer< double >.
- **AgilentEEsof::DComplexCircularBuffer**: Stores std::complex<double> objects (behaves like std::complex<double> *). Same as AgilentEEsof::CircularBuffer< std::complex < double > >.
- **AgilentEEsof::FloatCircularBuffer**: Stores float objects (behaves like float *). Same as AgilentEEsof::CircularBuffer< float >.
- **AgilentEEsof::FComplexCircularBuffer**: Stores std::complex<float> objects (behaves like std::complex<float> *). Same as AgilentEEsof::CircularBuffer< std::complex < float > >.
- **AgilentEEsof::FixedPointCircularBuffer**: Stores AgilenEEsof::FixedPoint objects (behaves like AgilenEEsof::FixedPoint *). Same as AgilentEEsof::CircularBuffer< AgilentEEsof::FixedPoint >. For more details about the SystemVue fixed point data type AgilentEEsof::FixedPoint see the section SystemVue FixedPoint Data Type.
- **AgilentEEsof::BoolMatrixCircularBuffer**: Stores BoolMatrix (Matrix<bool>) objects. Same as AgilentEEsof::CircularBuffer< AgilentEEsof::Matrix < bool > >.
- **AgilentEEsof::IntMatrixCircularBuffer**: Stores IntMatrix (Matrix<int>) objects. Same as AgilentEEsof::CircularBuffer< AgilentEEsof::Matrix < int > >.
- **AgilentEEsof::DoubleMatrixCircularBuffer**: Stores DoubleMatrix (Matrix<double>) objects. Same as AgilentEEsof::CircularBuffer< AgilentEEsof::Matrix < double > >.
- **AgilentEEsof::DComplexMatrixCircularBuffer**: Stores DComplexMatrix (Matrix<std::complex<double>>) objects. Same as AgilentEEsof::CircularBuffer< AgilentEEsof::Matrix < std::complex < double > > >.
- **AgilentEEsof::FloatMatrixCircularBuffer**: Stores FloatBoolMatrix (Matrix<float>) objects. Same as AgilentEEsof::CircularBuffer< AgilentEEsof::Matrix < float > >.
- **AgilentEEsof::FComplexMatrixCircularBuffer**: Stores FComplexMatrix (Matrix<std::complex<float>>) objects. Same as AgilentEEsof::CircularBuffer< AgilentEEsof::Matrix < std::complex < float > > >.

For more details about the SystemVue matrix data type AgilentEEsof::Matrix<T> see the section SystemVue Matrix Data Type. To make use of any of the **MatrixCircularBuffer** data types, the header file MatrixCircularBuffer.h needs to be included.

Note

- CircularBuffer< T > data types are only designed to be used as inputs/outputs and not for any other purpose.
- CircularBuffer< T > data types are the most efficient way to implement inputs/outputs; it is highly recommended that they are used instead of the built in C++ data types.
- The **[]** and **IsConnected()** must not be used outside *Run()* (users) method of your model.

SystemVue CircularBufferBus Data Types

The CircularBufferBus data types are the only way to implement a bus input or bus output. The bus inputs/outputs are shown as double arrow ports on the SystemVue schematic. The following CircularBufferBus data types are predefined and available for use as bus inputs/outputs (to make use of these data types, the header file MatrixCircularBuffer.h needs to be included):

- **AgilentEEsof::BoolCircularBufferBus**: Bus of AgilentEEsof::BoolCircularBuffer.
- **AgilentEEsof::IntCircularBufferBus**: Bus of AgilentEEsof::IntCircularBuffer.
- **AgilentEEsof::DoubleCircularBufferBus**: Bus of AgilentEEsof::DoubleCircularBuffer.
- **AgilentEEsof::DComplexCircularBufferBus**: Bus of AgilentEEsof::DComplexCircularBuffer.
- **AgilentEEsof::FloatCircularBufferBus**: Bus of AgilentEEsof::FloatCircularBuffer.
- **AgilentEEsof::FComplexCircularBufferBus**: Bus of AgilentEEsof::FComplexCircularBuffer.
- **AgilentEEsof::FixedPointCircularBufferBus**: Bus of AgilentEEsof::FixedPointCircularBuffer.
- **AgilentEEsof::BoolMatrixCircularBufferBus**: Bus of AgilentEEsof::BoolMatrixCircularBuffer.
- **AgilentEEsof::IntMatrixCircularBufferBus**: Bus of AgilentEEsof::IntMatrixCircularBuffer.
- **AgilentEEsof::DoubleMatrixCircularBufferBus**: Bus of AgilentEEsof::DoubleMatrixCircularBuffer.
- **AgilentEEsof::DComplexMatrixCircularBufferBus**: Bus of AgilentEEsof::DComplexMatrixCircularBuffer.
- **AgilentEEsof::FloatMatrixCircularBufferBus**: Bus of AgilentEEsof::FloatMatrixCircularBuffer.
- **AgilentEEsof::FComplexMatrixCircularBufferBus**: Bus of AgilentEEsof::FComplexMatrixCircularBuffer.

Using CircularBufferBus Data Types

The CircularBufferBus data types are the only way to implement a bus type input or output. The size of the Bus can be accessed using the **size_t GetSize()** method. An individual CircularBuffer can be accessed using the [] operator. To access the j^{th} data sample of the i^{th} input connected to the bus use input[i][j]. For example, input[0][2] can be used to access 3rd data sample (indexed by 2) in the first multi-rate input (indexed by 0) connected to the bus input. The outputs can be accessed similarly.

SystemVue Timed Circular Buffer

SystemVue provides **AgilentEEsof::TimedCircularBuffer<T>** for a timed model to access *time stamps* (sim) of the input (or output) data samples and to set sample rate and latency information. AgilentEEsof::TimedCircularBuffer is defined in \ModelBuilder\include\SystemVue\TimedCircularBuffer.h in the SystemVue installation directory. **AgilentEEsof::TimedCircularBuffer<T>** inherits from **AgilentEEsof::CircularBuffer<T>** to provide additional timing information using **AgilentEEsof::CircularBufferTime** class. The member methods of AgilentEEsof::TimedCircularBuffer are described as follows.

- **double GetTime(size_t iIndex, unsigned long long iCount) const**: Get the time stamp at the *iCount* th firing of the model and the *iIndex* th sample of the buffer. Use this method in TimedDFModel::Run() to get the time stamp of a particular sample.
- **bool SetSampleRate(double dSampleRate)**: Set the sample rate, *dSampleRate*, and the corresponding time step (*1/dSampleRate*) of the model's input (or output) represented by this circular buffer. Use this method in TimedDFModel::Setup(). Return false if *dSampleRate* is not greater than 0.
- **bool SetTimeStep(double dTimeStep)**: Set the time step, *dTimeStep*, and the corresponding sample rate (*1/dTimeStep*) of the model's input (or output) represented by this circular buffer. Use this method in TimedDFModel::Setup(). Return false if *dTimeStep* is not greater than 0.
- **void SetStartTime(double dStartTime)**: Set the start time of the output. Use this method in TimedDFModel::CalculateLatency(). See *Overriding Latency Calculation* (users).
- **double GetSampleRate() const**: Get the sample rate. This method can be used after TimedDFModel::Setup() is called.
- **double GetTimeStep() const**: Get the time step. This method can be used after TimedDFModel::Setup() is called.
- **double GetStartTime() const**: Get the start time. This method can be used after TimedDFModel::Setup() and TimedDFModel::CalculateLatency() are called.

SystemVue also provides **AgilentEEsof::TimedCircularBufferE<T>**, which inherits from **AgilentEEsof::CircularBufferE<T>** and provides similar timed circular buffer implementation for data types that have internal memory.

SystemVue Envelope Circular Buffer

AgilentEEsof::EnvelopeCircularBuffer inherits from **TimedCircularBuffer< EnvelopeSignal >** and uses a private member **double m_dFc** to store the characterization frequency associated with the envelope signal. The member methods of AgilentEEsof::EnvelopeCircularBuffer are described as follows.

- **EnvelopeCircularBuffer()** : Default constructor, the characterization frequency is default to 0.
- **double GetCharacterizationFrequency()** : Get characterization frequency

- **void SetCharacterizationFrequency(double dFc)** : Set characterization frequency.

typedef CircularBufferBusT<EnvelopeCircularBuffer> EnvelopeCircularBufferBus is also defined in \ModelBuilder\include\SystemVue\EnvelopeSignal.h for easy usage of envelope signal circular buffer bus.

> Analytic signal is naturally associated with timing information — it requires time stamp to obtain the real baseband form or to convert to another characterization frequency. As a result, EnvelopeCircularBuffer is designed to inherit from TimedCircularBuffer in order to access the timing information. For the same reason, models that use envelope signal are usually inherited from AgilentEEsof::TimedDFModel.

SystemVue FixedPoint Data Type

SystemVue provides AgilentEEsof::FixedPoint data type that is similar in computational behavior to SystemC TM 2.2 fixed point type based on IEEE Std. 1666 TM Language Reference Manual (LRM) . However, the actual API is modified to suit C++ modeling in SystemVue. The major differences in AgilentEEsof::FixedPoint API and SystemC TM 2.2 fixed point data type API are described below:

- The AgilentEEsof::FixedPoint data type can be configured as both signed (2's complement) and unsigned.
- The FixedPointParameters can be changed using SetParameter method of AgilentEEsof::FixedPoint any time, whereas in SystemC TM 2.2 , the scfx_params cannot be modified after the construction of sc_fix or sc_ufix. This is needed because fixed point parameters are dependent on user specified values through model parameters.
- Unlike sc_fix and sc_ufix, the AgilentEEsof::FixedPoint has a default constructor and a copy constructor. To specify AgilentEEsof::FixedPointParameters, you must call a SetParameter method.
- Unlike sc_fix and sc_ufix, the AgilentEEsof::FixedPoint provides only bit references and not sub-references.

The computational behavior such as overflow, quantization, effect of integer word length (which could be negative or larger than word length) is similar to that of SystemC TM 2.2 . SystemVue also provides an arbitrary precision fixed point data type **AgilentEEsof::FixedPointValue**. The data stored in AgilentEEsof::FixedPointValue does not lose bit-width precision and/or location of binary point i.e. no overflow or quantization handling is performed on an object of AgilentEEsof::FixedPointValue.

> **Warning**
> An object of AgilentEEsof::FixedPoint and AgilentEEsof::FixedPointValue cannot be used as an input or an output, use AgilentEEsof::FixedPointCircularBuffer or AgilentEEsof::FixedPointCircularBufferBus instead.

AgilentEEsof::FixedPoint Constructors

The AgilentEEsof::FixedPoint provides

- A default constructor which sets the fixed point properties as follows
  - Word Length (wl) = 32
  - Integer Word Length (iwl) = 32
  - Sign = AgilentEEsof::FixedPointEnums::TWOS_COMPLEMENT
  - SaturationBits = 0
  - QuantizationMode = AgilentEEsof::FixedPointEnums::TRUNCATE
  - OverflowMode = AgilentEEsof::FixedPointEnums::WRAP
- A copy constructor

AgilentEEsof::FixedPoint Mutators

The AgilentEEsof::FixedPoint provides following mutators to set fixed point parameters

void setParameters(FixedPointEnums::Sign eSign, FixedPointEnums::QuantizationMode qm=FixedPointEnums::TRUNCATE, FixedPointEnums::OverflowMode om=FixedPointEnums::WRAP, int nb=0);

where

- **eSgin** could be FixedPointEnums::TWOS_COMPLEMENT OR FixedPointEnums::UNSIGNED .
- **qm** specifies the quantization mode, possible values are. Note that "FixedPointEnums" is a nested namespace inside AgilentEEsof namespace (AgilentEEsof::FixedPointEnums)
  - FixedPointEnums::ROUND - Rounding to Plus infinity.
  - FixedPointEnums::ROUND_ZERO - Rounding to Zero.
  - FixedPointEnums::ROUND_MINUS_INFINITY - Rounding to Minus infinity.
  - FixedPointEnums::ROUND_INFINITY - Rounding to infinity.
  - FixedPointEnums::ROUND_CONVERGENT - Convergent rounding.
  - FixedPointEnums::TRUNCATE - Truncation.
  - FixedPointEnums::TRUNCATE_ZERO - Truncation to zero.
- **om** specifies the overflow mode, possible values are. Note that "FixedPointEnums" is a nested namespace inside AgilentEEsof namespace (AgilentEEsof::FixedPointEnums)
  - FixedPointEnums::SATURATE - Saturation
  - FixedPointEnums::SATURATE_ZERO - Saturation to Zero.
  - FixedPointEnums::SATURATE_SYMMETRICAL - Symmetrical saturation.
  - FixedPointEnums::WRAP - Wrap-around.
  - FixedPointEnums::WRAP_SIGN_MAGNITUDE - Sign magnitude wrap-around.
- **nb** is used to provide number of saturation bits for FixedPointEnums::WRAP and FixedPointEnums::WRAP_SIGN_MAGNITUDE Overflow modes.

void setParameters(int wl, int iwl, FixedPointEnums::Sign eSign, FixedPointEnums::QuantizationMode qm=FixedPointEnums::TRUNCATE, FixedPointEnums::OverflowMode om=FixedPointEnums::WRAP, int nb=0);

where

- **wl** specifies the word length.
- **iwl** specifies the integer word length.
  Other parameters have the same meaning as mentioned above.

void setParameters(const FixedPointParameters & cParams);

where **cParams** is an object of **AgilentEEsof::FixedPointParameters**. The **AgilentEEsof::FixedPointParameters** is used to hold all fixed point parameter information. Please look at the **FixedPointParameters.h** file under **<SystemVue Install Directory>\ModelBuilder\inlcude** directory to use this class.

AgilentEEsof::FixedPoint Bit Selection Operator/Method

- **The [] Operator:** The operator [i] returns a reference (FixedPointBitRef) to i th bit in the corresponding FixedPoint object. It is to be noted that value of index **i** can be negative to access fractional bits. For example myFix[-2] will return the bit reference of 2 nd fractional bit to the right of the **point** in object myFix, and myFix[3] points to the 4 th integer bit to the left of the **point**. Except the indexing scheme specific to the FixedPoint, the [] can be used exactly the same manner as [] operator an array type.
- **FixedPointBitRef bit(int i);** The bit(i) method returns a reference (FixedPointBitRef) to i th bit in the corresponding FixedPoint object. The indexing scheme is same as [] operator and the value of index can be negative for fractional bits.

AgilentEEsof::FixedPoint Explicit Conversion Methods

- **short to_short() const;** Explicit conversion to **short**.
- **unsigned short to_ushort() const;** Explicit conversion to **unsigned short**.
- **int to_int() const;** Explicit conversion to **int**.
- **unsigned intto_uint() const;** Explicit conversion to **unsigned int**.
- **long to_long() const;** Explicit conversion to **long**.
- **unsigned long to_ulong() const;** Explicit conversion to **unsigned long**.

- **float to_float() const;** Explicit conversion to **float**.
- **double to_double() const;** Explicit conversion to **double**.
- **const std::string to_dec() const;** Explicit conversion to std::string in decimal format
- **const std::string to_bin() const;** Explicit conversion to std::string in binary format
- **const std::string to_oct() const;** Explicit conversion to std::string in octal format
- **const std::string to_hex() const;** Explicit conversion to std::string in hexa-decimal format

> **Warning**
> Implicit conversion to any of the above mentioned data type is not supported. You must use explicit conversion method above for conversions.

AgilentEEsof::FixedPoint Query Methods

- **bool is_neg() const;** Returns true if negative
- **bool is_zero() const;** Returns true if Zero
- **bool quantization_flag() const;** Returns true if quantization flag is set. This means that last assignment operator has caused quantization.
- **bool overflow_flag() const;** Returns true if overflow flag is set. This means that last assignment operator has caused oveflow.
- **int wl() const;** Returns word length.
- **int iwl() const;** Returns integer word length.
- **FixedPointEnums::QuantizationMode q_mode() const;** Returns quantization mode. The return mode has one of the values specified in SetParameter method above. Also see **FixedPointEnums.h** as reference.
- **FixedPointEnums::OverflowMode o_mode() const;** Returns Overflow mode. The return mode has one of the values specified in SetParameter method above. Also see **FixedPointEnums.h** as reference.
- **FixedPointEnums::Sign sign() const;** Returns sign, either AgilentEEsof::FixedPointEnums::TWOS_COMPLEMENT or AgilentEEsof::FixedPointEnums::UNSIGNED.
- **int saturationBits() const;** Returns number of saturation bits used for FixedPointEnums::WRAP and FixedPointEnums::WRAP_SIGN_MAGNITUDE Overflow modes.
- **const FixedPointParameters & getParameters() const;** Returns an object of FixedPointParameters.
- **AgilentEEsof::FixedPoint Assignment Operators:** The **=**, ***=**, **/=**, **+=**, and **-=** operators are supported for **int**, **unsigned int**, **long**, **unsigned long**, **double**, **const FixedPointValue**, and **const FixedPoint**. The **<<=**, and **>>=** operaors can be used for left, and right shift respectively, The value to the right of operator specifies the amount of shift.
- **Bitwise Binary Operators:** The three bitwise binary operators **|** (OR), **&** (AND), and **^** (XOR) operators are supported bitween two FixedPoint objects. These operators does not check that FixedPointParameters are same for both inputs, it is the users responsibility to check for any FixedPointParameters consistency between two inputs if needed.
- **Binary Operators:** The binary operators ***** (multiplication), **/** (division), **+** (addition), and **-** (subtraction) is supported between an object of FixedPoint and one of the listed data types **FixedPoint**, **FixedPointValue**, **int**, **unsigned int**, **long**, **unsigned long**, and **double**. These binary operators returns an object of type **FixedPointValue** which is arbitrary precision fixed point representation to avoid any loss of information.

AgilentEEsof::FixedPointValue

SystemVue also provides an arbitrary precision fixed point data type **AgilentEEsof::FixedPointValue**. The data stored in AgilentEEsof::FixedPointValue does not lose bit-width precision and/or location of binary point i.e. no overflow or quantization handling is performed on an object of AgilentEEsof::FixedPointValue. The objects of FixedPointValue and FixedPoint works seamlessly for all binary operations except bitwise operations such as AND (&), OR(|), XOR(^) which works only with FixedPoint type. The major difference between FixedPoint and FixedPointValue are as follows.

- An object of FixedPointValue stores data without performing overflow and/or quantization whereas FixedPoint performs quantization/overflow handling with each assignment operator call.
- An object of FixedPointValue cannot be used to perform bitwise operations such as &, |, and ^; an object of FixedPoint needs to be used for this purpose.
- An individual bit in a FixedPointValue cannot be accessed whereas it can be accessed in an object of FixedPoint type.

One recommended place to use FixedPointValue is as an accumulator data for internal computation, for example in case of an adder with bus input it is better to accumulate the sum using FixedPointValue and at the end assign it to the corresponding output. This will cause overflow/quantization handling at the output only once.

SystemVue Matrix Data Type

SystemVue provides a matrix data type AgilentEEsof::Matrix<T>, which implements 2-dimensional matrices (1-dimensional matrices can be defined by setting the number of rows or columns to 1). The matrix data type is implemented as a templated class so that matrices of different data types can be defined. Template instantiations of this class for the most commonly used types (bool, int, float, double, std::complex<float>, std::complex<double>) have been predefined for ease of use. The AgilentEEsof::Matrix<T> class is a very light weight class; it only provides some very basic matrix operations. Its intend is to facilitate efficient data movement between models and some basic matrix math operation. Its intend is not to provide a full featured matrix class with a rich set of matrix math operations. The table below summarizes the methods defined in this class:

Method	Description
Matrix()	Constructor - creates empty matrix.
Matrix(size_t nRows, size_t nCols)	Constructor - creates uninitialized nRows x nCols matrix.
Matrix(const Matrix & matrix)	Copy Constructor.
~Matrix()	Destructor.
void Resize(size_t nRows, size_t nCols)	Resize matrix to nRows x nCols.
size_t NumRows() const	Return the number of rows.
size_t NumColumns() const	Return the number of columns.
size_t NumElements() const	Return the number of matrix elements.
void SetMaxElements(size_t iMaxElements)	Set the maximum number of elements the matrix can hold.
bool Zero()	Set all elements to zero.
bool Zero(Matrix* pReference)	Resize based on dimensions of a reference matrix and set all elements of resized matrix to zero.
bool operator == (const Matrix & matrix) const	Return TRUE if this matrix is equal to another one.
bool operator != (const Matrix & matrix) const	Return TRUE if this matrix is not equal to another one.
Matrix& operator = (const Matrix& matrix)	Assignment operator (copy contents of right hand side operand to left hand side operand).
template <typename T2> void CopyFrom(const T2* pData, size_t iSize)	Copy iSize elements from address pData to this matrix.
template <typename T2> void CopyTo(T2* pData, size_t iSize) const	Copy the first iSize matrix elements to address pData.
T& operator() (size_t iRow, size_t iCol)	Return a reference to the matrix element at row iRow and column iCol.
T operator() (size_t iRow, size_t iCol) const	Return the matrix element at row iRow and column iCol.
T& operator() (size_t iIndex)	Return a reference to the iIndex matrix element (elements stored in column major form).
T operator() (size_t iIndex) const	Return the iIndex matrix element (elements stored in column major form).
Matrix& operator-()	Negate matrix.
template<typename S> Matrix& operator+= (S scalar)	Add scalar to each matrix element.
template<typename M> Matrix& operator+= (const Matrix<M>& matrix)	Matrix addition.
template<typename S> Matrix& operator-= (S scalar)	Subtract scalar from each matrix element.
template<typename M> Matrix& operator-= (const Matrix<M>& matrix)	Matrix subtraction.
template<typename S> Matrix& operator*= (S scalar)	Multiply each matrix element with a scalar.
template<typename T2> Matrix& operator*= (const Matrix<T2>& matrix)	Matrix multiplication.
bool diagonal(T data)	Make this matrix a diagonal one with all diagonal elements set to data.
bool identity()	Make this matrix an identity one.
T* GetBuffer()	Get access to the internal storage array. Matrix elements are stored in column major form.
const T* GetBuffer() const	Get access to the internal storage array (const version). Matrix elements are stored in column major form.
void Swap(Matrix* pMatrix)	Swap contents with another matrix.

In addition, the following matrix related functions are defined in the AgilentEEsof namespace:

Function	Description
template <typename M1, typename M2, typename M3> Matrix<M1> operator + (const Matrix<M2> &mx1, const Matrix<M3> &mx2)	Return sum of matrices mx1 and mx2.
template <typename M1, typename M2, typename M3> Matrix<M1> operator + (const Matrix<M2> &mx1, const M3 &mx2)	Return sum of matrix mx1 and scalar mx2.
template <typename M1, typename M2, typename M3> Matrix<M1> operator + (const M3 &mx2, const Matrix<M2> &mx1)	Return sum of scalar mx2 and matrix mx1.
template <typename M1, typename M2, typename M3> Matrix<M1> operator - (const Matrix<M2> &mx1, const Matrix<M3> &mx2)	Return difference of matrices mx1 and mx2 (mx1 - mx2).
template <typename M1, typename M2, typename M3> Matrix<M1> operator - (const Matrix<M2> &mx1, const M3 &mx2)	Return matrix mx1 minus scalar mx2.
template <typename M1, typename M2, typename M3> Matrix<M1> operator - (const M3 &mx2, const Matrix<M2> &mx1)	Return scalar mx2 minus matrix mx1.

For more details see comments in the shipped header file Matrix.h.

SystemVue Envelope Signal Data Type

A real-valued signal x(t) can be represented in the form of analytic signal $x_a(t) = x_c(t) \exp(j 2 \pi f_c t)$. In this representation, $x_c(t)$ is defined as the **complex envelope** of x(t), and f_c is the **characterization frequency** associated with the complex envelope. Complex envelope $x_c(t)$ is a complex-valued signal. It can be expressed as $x_c(t) = x_i(t) + j x_q(t)$, where $x_i(t)$ and $x_q(t)$ are both real-valued signals and are referred to as the **in-phase component** the **quadrature component** of x(t). Using this form, the real signal can be expressed as $x(t) = \mathrm{Real}\{x_a(t)\} = x_i(t) \cos(2 \pi f_c t) - x_q(t) \sin(2 \pi f_c t)$.

In SystemVue, a modulated passband signal is usually represented as a time varying **complex envelope** signal $x_c(t)$ associated with a constant positive f_c in the **envelope signal data type**. The benefit of using SystemVue **envelope signal** to represent modulated signal is that the sample rate needed to fully represent a complex envelope signal can be in the order of the information bandwidth, which is in general orders of magnitude smaller than the sample rate required for direct real signal representation.

SystemVue **envelope signal** can represent EITHER a **real signal** x(t) OR an **analytic signal** $x_c(t) \exp(j 2 \pi f_c t)$ (which is equivalent to a **complex envelope** signal $x_c(t)$ with associated constant f_c). The choice of representation is based on the characterization frequency f_c associated with the envelope signal.

If $f_c = 0$, SystemVue treats the envelope signal as a real signal x(t). If $f_c > 0$, SystemVue treats the envelope signal as an analytic signal $x_a(t) = x_c(t) \exp(j 2 \pi f_c t)$ (or equivalently a complex envelope signal $x_c(t)$ with associated f_c).

SystemVue currently does not support $f_c < 0$.

For more detailed discussion about SystemVue envelope signal, we refer the users to *Envelope Signal* (sim).

Envelope Signal Type

SystemVue provides **AgilentEEsof::EnvelopeSignal** to represent the envelope signal data type introduced above and provides **AgilentEEsof::EnvelopeCircularBuffer** to access the characterization frequency associated with the envelope signal. AgilentEEsof::EnvelopeSignal and AgilentEEsof::EnvelopeCircularBuffer are defined in \ModelBuilder\include\SystemVue\EnvelopeSignal.h in the SystemVue installation directory.

AgilentEEsof::EnvelopeSignal uses a private member **std::complex<double> m_cxSignal** to store the value of a real sample or a complex envelope sample. The member methods of AgilentEEsof::EnvelopeSignal are described as follows.

- **EnvelopeSignal()** : Default constructor, m_cxSignal is default to 0.
- **EnvelopeSignal(const std::complex<double>& cx)** : Convert constructor, m_cxSignal is set to *cx*.
- **double real() const** : Get real part. If the associated characterization frequency is 0, use this method to get the real baseband signal value.
- **double imag() const** : Get imaginary part.
- **std::complex<double> complex() const** : Get complex value. If the associated characterization frequency is greater than 0, use this method to get the complex

envelope I-Q value.

- **EnvelopeSignal& operator = (const std::complex<double>& cx)** : Assignment operator for std::complex<double>. Use this method to assign complex envelope I-Q value to the EnvelopeSignal.
- **EnvelopeSignal& operator = (const double& d)** : Assignment operator for double. Use this method to assign real baseband value to the EnvelopeSignal.
- **double ConvertToReal(double dFc, double dTime) const** : Convert complex envelope I-Q representation to real baseband signal. *dFc* is the characterization frequency. *dTime* is the time stamp of the EnvelopeSignal sample, which can be obtained from TimedCircularBuffer::GetTime(size_t iIndex, unsigned long long iCount). Use this method only if the associated characterization frequency is greater than 0.
- **std::complex<double> ConvertToNewFc(double dFc, double dNewFc, double dTime) const** : Convert complex envelope I-Q value characterized at *dFc* to the equivalent I-Q representation at characterization frequency *dNewFc* and return the converted complex envelope I-Q value. *dFc* is the characterization frequency associated with the envelope signal. *dNewFc* is the new characterization frequency. *dTime* is the time stamp of the EnvelopeSignal sample, which can be obtained from TimedCircularBuffer::GetTime(size_t iIndex, unsigned long long iCount). Use this method only if the associated characterization frequency is greater than 0.

Sub-Network Models

A sub-network model is used to abstract a model or group of models to something easier to use and manage from a users perspective. This type of model hides implementation details that may confuse the user or distract from the readability of a simulation topology.

For example, a user may want to simulate the effects of a non-linear filter. Models exist for filters and non-linear blocks. However, there is no non-linear filter model. A new sub-network model can be created out of the two existing models. The parameters from these two models can be abstracted to only reveal parameters that user would be interested in entering for this type of sub-network model.

The abstraction of the Sub-Network model in SystemVue is really an object called a design. The Sub-Network model is really a design object with the following attributes:

- PartList
- Schematic
- Equations
- Parameters
- Notes

All of these attributes are interrelated except for the Notes which serve as documentation or help for the Sub-Network model.

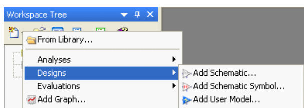
Contents

Creating a Parameterized Sub-Network Model

There are two different ways to to create a sub-network model in SystemVue. One is by clicking on the New Item button ( ) on the Workspace Tree toolbar. The other is by right clicking on a folder in the workspace tree.

Method 1 - **Clicking on the New Item Button**

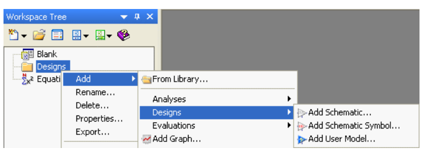
1. Click the New Item button ( ) on the Workspace Tree toolbar
2. Select the 'Designs >' submenu
3. Now select 'Add User Model...'
4. This model will be added under the folder that last selected in the workspace tree



Or

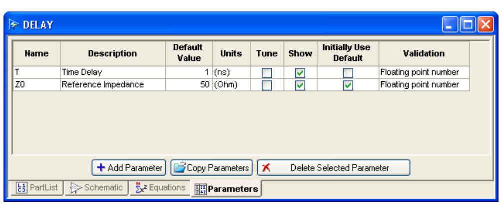
Method 2 - **Right Clicking on a Workspace Folder**

1. Right click on a folder in the workspace tree to bring up the right click menu.
2. Select the 'Add >' submenu.
3. Select the 'Designs >' submenu
4. Now select 'Add User Model...'
5. The design will be added under the folder that was initially right clicked



Note
If you create the new design in the wrong folder, simply drag it to the folder of interest.

Entering User Parameters



1. Add new parameters by clicking the Add Parameter button.
2. Copy selected parameters (from the parts in the design) into the list by clicking the Copy Parameters button. A selection dialog box is displayed. Select individual parameters (to copy from the base design) by placing a checkmark beside each parameter you wish to copy. Then click OK.
   Tip: This is the recommended way of adding parasitics (etc.) to an existing part (a "user model").
3. Delete unwanted entries with the Delete Selected Parameter button.

- **Name** - The name of the parameter.
- **Description** - A short description of the parameter
- **Default Value** - The normal, standard value for this parameter
- **Units** - The units-of-measure for this parameter
- **Tune** - Is it normally tuned?
- **Show** - Is it normally shown on a schematic?
- **Initially Use Default** - Should the Default value be used when the part is placed on a schematic?
- **Validation** - Usage rules that determine if a parameter value is valid and in-range. See details below.
- **Hide Condition** - Dependence of the activity and visibility of the parameter on values of other parameters of the design. See details below.

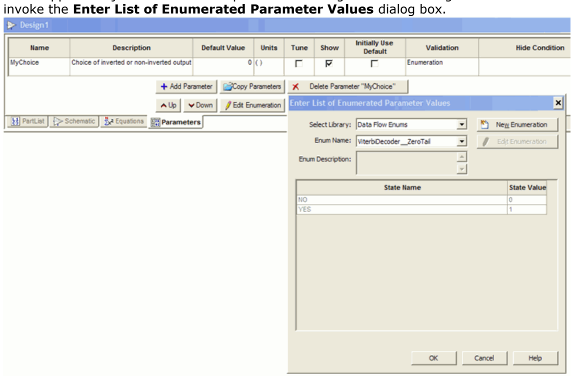
Validation Types

Type	Comment
Floating point number	1.0, 1e-6, etc. are valid entries
Warn if negative	Posts a warning if the value is < 0
Warn if non-positive	Posts a warning if the value is < 1
Positive integer	Only numbers like 1, 2, 3, ... are allowed
<None>	No validation will be performed
Text	The parameter is a string; any text is valid
Warning	Always generates a warning
Error if negative	Posts an error if the value is < 0
Error if non-positive	Posts an error if the value is < 1
Error	Always generates an error
Filename	Brings up a browse button for file selection as well option for manual a text entry
Integer	Any integer value
Complex number	Complex number in RI MathLang syntax, e.g. X + j*Y. Real and integer values supported by default
Integer array	Fully defined MxN array of integers with comma delimited columns and semi-colon delimited rows
Floating point array	Fully defined MxN array of integer or real numbers with comma delimited columns and semi-colon delimited rows
Complex array	Fully defined MxN array of integer, real or complex numbers with comma delimited columns and semi-colon delimited rows
Enumeration	Allows definition of arbitrary user-defined labels and options for assigning values to the parameter of interest

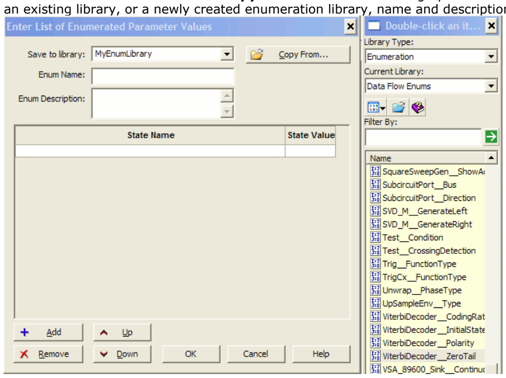
Note
An array parameter may be specified as a scalar number without any delimiters as in MyArray0=2.345. It may be a one-dimensional vector as in MyArray1=[2, 3, -4].
Two-dimensional arrays are specified row over column as MyArray2=[1, 2, 3; 4, 5, 6] where the first three parts form the first row.
Higher dimensions are created by appending nested versions of 2-D representations separated by semi-colons e.g.
MyArray3=[[1, 2; 3, 4; 5, 6]; [-1, -2; -3, -6; -5, -4]].
This is a 3-D array consisting of 2 separate 3x2 2-D arrays such that matrix size is 2x3x2.

Using enumerated parameters

The process of defining an enumerated parameter starts with the selection of this validation type followed by selection of the context sensitive **Edit Enumeration** button which appears adjacent to the other parameter editing buttons. Clicking this button will invoke the **Enter List of Enumerated Parameter Values** dialog box.

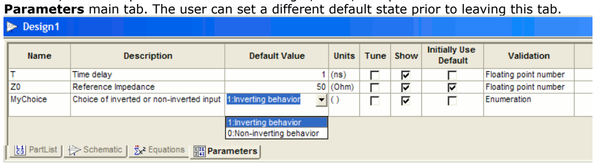


It is possible to choose a pre-defined enumeration template by selecting the library and enumeration name. Customizations can be performed by clicking the **New Enumeration** button which is transformed into a **Copy From ...** button to allow graphical selection from an existing library, or a newly created enumeration library, name and description.



Names and states of manually created enumerations or modifications of existing enumerations can be directly entered into the table and organized using the **Add**, **Remove**, **Up** and **Down** buttons.

Upon accepting the enumeration list, the corresponding drop-down menu is created under the **Default Values** column for this parameter in the main **Parameter** tab. Note that the first entry of the enumeration table will be treated as the initial default value regardless of the state number associated with it. In this example, the first entry was *1:Inverting Behavior*, which despite its state number being 1, not 0, was picked as the default in the **Parameters** main tab. The user can set a different default state prior to leaving this tab.



Setting Hide Condition

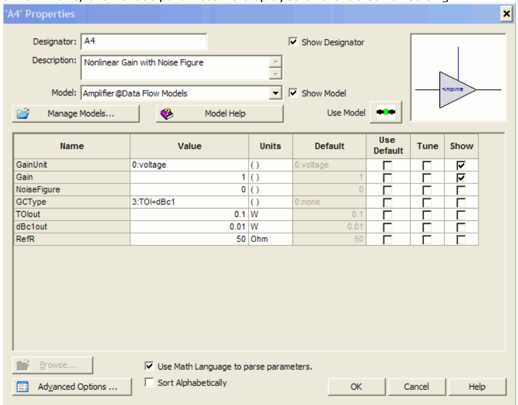
The final column of the Parameter entry table allows the user to set up boolean expressions for hiding and deactivating parameters based on the values of other parameters. The boolean expressions entered in this column must be written using MathLang syntax. If the expression evaluates to TRUE the parameter will be hidden and deactivated (ignored for simulation purposes). If the expression evaluates to FALSE the parameter will be visible and active (it will be used for simulation purposes).

One example is shown in the *Amplifier* part of the *Algorithm Design* library. This built-in component has a total of 11 parameters as shown in the model view which can be imported from the library into any workspace.

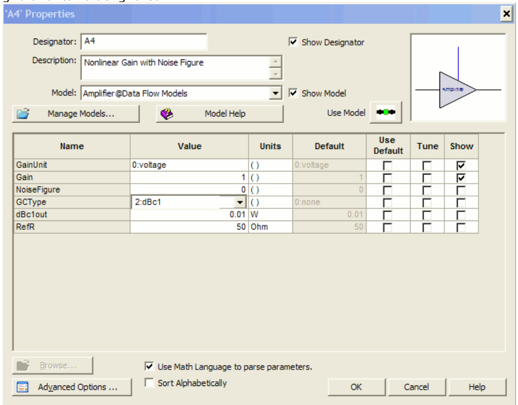
Name	Description	Default Value	Units	Validation	Hide Condition
GainUnit	Gain unit for the Gain para	0:voltage	( )	Enumeration	
Gain	Gain with units defined by	1	( )	Floating point number	
NoiseFigure	Input noise figure in dB	0	( )	Floating point number	
GCType	Gain compression type	0:none	( )	Enumeration	
TOIout	Output third order intercept	0.1	W	Floating point number	(GCType ~= 1) && (GCType ~= 3) && (GCType ~= 4) && (GCType ~= 6)
dBc1out	Output 1 dB gain compressi	0.01	W	Floating point number	(GCType ~= 2) && (GCType ~= 3) && (GCType ~= 5) && (GCType ~= 6)
PSat	Saturation power	0.032	W	Floating point number	(GCType ~= 4) && (GCType ~= 5) && (GCType ~= 6) && (GCType ~= 7)
GCSat	Gain compression at satura	3	( )	Floating point number	(GCType ~= 4) && (GCType ~= 5) && (GCType ~= 6)
RappS	Rapp nonlinearity smoothne	3	( )	Integer	GCType ~= 7
GComp	Array of triple values for In	[0, 0, 0]	( )	Floating point array	(GCType ~= 8) && (GCType ~= 9)
RefR	Reference resistance	50	Ohm	Floating point number	

Observe that the parameters *TOIout*, *dBc1out*, *PSat*, *GCSat*, *RappS* and *GComp* all have Hide Conditions defined based on the value of the *GCType* paramater. For instance, *TOIout* is to be hidden and its assigned value ignored if *GCType* is NOT in the set {1, 3, 4, 6}, in which case the Hide Condition for *TOIout* evaluates to FALSE. The corresponding behavior can be observed when placing an instance of the part on a schematic and double clicking on it.

When *GCType* is selected to be a member of the above mentioned set, e.g. to 3:TOI+1dBc, the *TOIout* parameter is displayed and enabled for editing.



Setting *GCType* to a non-member of the above set, e.g. 2:1dBc, results in a TRUE value for the Hide Condition and therefore the parameter *TOIout* is hidden from the parameter grid and its value ignored.



> **Note**
> Defining **Hide Conditions** refers strictly to the table view of parameters and not the visibility of selected parameters on the schematic. Parameters that are hidden by condition are barred from schematic display even if they had the **Show** button checked prior to concealment.

As you simulate this design in the workspace, the default parameter values will be used in the simulation. When you use this design as a model in a part, the part parameters override these default parameter values.

Roles of Sub-Network Model Attributes

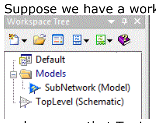
- **Parameters** - These are the parameters the user sees when entering values for this model.
- **Equations** - These are commonly used to manipulate data entered by the users to a format needed by models that appear in the schematic
- **Schematic** - This shows how existing models are visually and electrically connected together and their relationships with each other. The model parameters in the schematic can also use the top level parameters as well as any variable created in the equation block.
- **PartList** - This shows connectivity and part information in a table format.
- **Notes** - This is used for documentation or help for this sub-network.

Run-time Hierarchy - How Parameters get passed

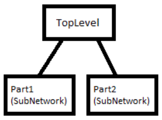
When a simulation is run, a model tree is instantiated that corresponds to the topology of the network you are simulating. This is called the *run-time hierarchy*. In contrast, when you are editing designs in the workspace, you are working in *design-time*. The difference will become apparent shortly.

Each part in your top level design references a model, and an instance of that model is created and set as a "child" of the top-level design when a simulation is run. If one of these children corresponds to a subnetwork model, then each model inside the subnetwork design is instantiated as well, recursively. It is easy to see why this sort of instantiation is necessary - you can have two parts in your top-level design that point to the same model, and they may have different values for their parameters.

Suppose we have a workspace as shown here (ie. this is the design-time hierarchy):



and suppose that TopLevel contains 2 instances of SubNetwork, ie. TopLevel has 2 parts called Part1 and Part2 whose models are both "SubNetwork". When you run a simulation on TopLevel, the following run-time model hierarchy is constructed:



Note that the Equations and Parameters of TopLevel are visible to the model instances of Part1 and Part2, but only at run-time!

It is important to note that when you are looking at the design called SubNetwork (ie. in design-time), and in its schematic you are using parameters defined in the Parameters tab of SubNetwork, the values you see at design-time will correspond to the "Default" values of the parameters as defined in the Parameters tab. This is because you are editing the Model called SubNetwork, but that model can be instantiated many times in your top-level network, and each instance can have different values for the parameters. Since you are editing the design-time model, it has no way of knowing what the values passed to it will be at run-time, and thus just shows the default values that are defined at design-time.

SystemVue 2007 APG DLL Import

You can import SystemVue 2007 MetaSystem designs as *Sub-Network Models* (users) (without the schematic) if you have the ability to create Automatic Program Generation (APG) DLLs.

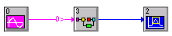
SystemVue 2007 APG Option **requires** a compatible Microsoft C compiler.

SystemVue 2007 MetaSystems

MetaSystems are the SystemVue 2007 mechanism for incorporating hierarchy into a design. For more information on MetaSystems please see *SystemVue 2007 User's Guide*. This section provides a very brief overview of MetaSystems as needed for import of your designs into SystemVue.

Creating a MetaSystem

To create a MetaSystem, click/drag the mouse to outline the tokens to be included in the new MetaSystem and then select *Tokens|Create MetaSystem* from the menu or click the *Create MetaSystem* button on the toolbar. The selected subsystem will be represented by a single MetaSystem token like token 3 on the picture. As you can see, it has become an equivalent of a sub-network with one input port and one output port.



When preparing a subsystem for import into SystemVue you need to leave stimulating sources and all the sinks out of the MetaSystem. Those connections will become ports which will allow you to place it within your SystemVue design. Of course, the whole MetaSystem could be a signal source in which case it would only have output connections.

Viewing and Saving a MetaSystem

Double click on a MetaSystem to enter the MetaSystem Window, where you can change connections and token parameters within the MetaSystem and add tokens, including new I/O tokens. To return to the main design, select *File|Return to System Level* from the menu or click the corresponding toolbar button.

MetaSystems are automatically saved with the parent system file (svu), and may also be saved to a separate file.

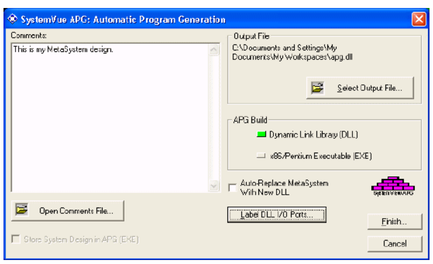
Building a SystemVue 2007 APG DLL

An APG DLL is a specialized SystemVue 2007 User Code DLL that is automatically generated from a MetaSystem. It contains one function with no adjustable parameters (although it could incorporate some globally linked tokens).

Only **connected** MetaSystem inputs and outputs are translated into the APG inputs and outputs. Therefore you must connect even the optional inputs and outputs before you create an APG DLL - if you want those inputs and outputs to be available in the resulting model.

APG Setup

Select *Tools|Auto Program Generation (APG)|Build MetaSystem DLL* from the menu and then click on the MetaSystem token. The following APG dialog window will appear.



- The *Comments* field is helpful for annotating your APG. The comments can be entered directly or imported from a text file.
- Click on the *Select Output File* button to select the name and folder for the APG DLL.
- Make sure to **uncheck** the *Auto Replace MetaSystem* checkbox.
- Click on the *Label DLL I/O Ports* button to identify the input and output connections. By default the labels indicate which tokens within the main design they are connected to.
- Click *Finish* to begin the build process. If successful, at the end you will see a message with the location of your APG DLL.

  If you forgot to uncheck the *Auto Replace MetaSystem* checkbox, the APG will replace your original MetaSystem. Select *Edit|Undo* from the menu to restore it.

  Very rarely you might see APG fail with a message "Cannot create APG SVA file." To troubleshoot, launch APG Setup again and click on the *Select Output File* button to select a different name for your APG. If the APG must have the same name, you need to save your system, then exit and restart SystemVue 2007.

Supported C Compilers

SystemVue 2007 APG Option requires a compatible Microsoft C compiler. Two supported compilers are

- Microsoft Visual Studio C++ .NET 2003 Professional Edition
- Microsoft Visual Studio C++ 2005 Professional Edition

Other compilers that can be used are

- Microsoft Visual C++ 2008 Express Edition
- Microsoft Visual Studio C++ 2008 with SP1

These compilers require the use of *Custom APG Build* as described below.

Custom APG Build

If you create a batch file named *apgbuild.bat* within your SystemVue 2007 installation folder, that batch file will be executed to create APG. This may be useful for customising the build or using a new compiler.

- The C source files are named **~apgtmp.c** and **~apgtmp1.c**. They are not human readable.
- The module definition file is named **~apgtmp.def**.
- The two APG libraries are named **ApgLibPC.lib** and **ApgUtlPC.lib**. They were created using Microsoft Visual C++ .NET, which limits the available linking options.
- The DLL being built should be named **~apgtmp.dll**.
- You may want to append the compiler output to **APGBUILD.LOG**.

In addition to setting the enviroment for the compiler, you may need to add include directories using /I option and list additional libraries for the linker. Please see the *Microsoft Visual C++ User's Guide* for details on using the command line compiler.

For example, this script works with Microsoft Visual C++ 2008 Express Edition.

```
call "C:\Program Files\Microsoft Visual Studio 9.0\VC\vcvarsall.bat"
cl 1>~apgtmp.out ~apgtmp1.c ~apgtmp.c /nologo /MT /O2 ^
/link /machine:ix86 /subsystem:windows /DLL /DEF:~apgtmp.def /OUT:~apgtmp.dll ApgLibPC.lib
ApgUtlPC.lib user32.lib
type ~apgtmp.out >>APGBUILD.LOG
echo +++ END OF LOG +++ >>APGBUILD.LOG
exit
```

Importing a SystemVue 2007 APG DLL into SystemVue

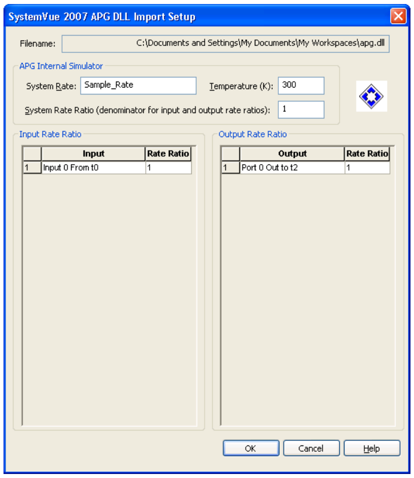
Different Simulation Engines

SystemVue and SystemVue 2007 have different simulation engines. SystemVue is a *data flow simulator* (sim), while SystemVue 2007 is a time based simulator. In order to translate a SystemVue 2007 subsystem into SystemVue model some additional information is required - you need to compute integer rate ratios between the system rate and different I/O token rates in the MetaSystem.

Computing rate ratios for a multi-rate system can be a challenge. A *Math Language* (users) script can assist you with this task.

SystemVue 2007 APG DLL Import Setup dialog

To import an APG DLL into SystemVue select *File|Import|SystemVue 2007 APG* from the menu. You will be prompted for a DLL file name. After selecting the APG DLL file you will see a SystemVue 2007 APG DLL Import Setup dialog.

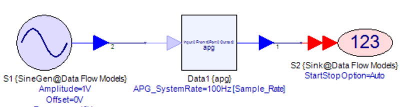


- *System Rate* is the SystemVue 2007 system sample rate. This is the sample rate of the time based simulator that runs inside the model. The default value is the variable *Sample_Rate* which represents the sample rate of the data flow simulator.
- The multi-rate properties of the subsystem must be expressed as integer ratios. Therefore an integer is assigned to the System Rate and each of the inputs and outputs of the subsystem. For instance, if the token rate of the MetaSystem input is the same as the system sample rate but the token rate of the output is 1/3 of the system sample rate, then the *System Rate Ratio* and the input *Rate Ratio* could both be 3, and thus the output *Rate Ratio* would be 1. Note that the rate ratios can be entered as formulas or variables computed using a *Math Language* (users) script.
- *Temperature* is only used if your subsystem contains tokens dependent on thermal noise.

After you click *OK* the APG *sub-network model* (users) will be placed on the *Workspace Tree* (users).

Using the APG Sub-Network Model

The SystemVue 2007 APG *sub-network model* (users) can be used as a part in your SystemVue design. Just drag it onto the schematic from the *Workspace Tree* (users).



You can also create a library of these models - simply right click and select *Copy To* from the popup menu.

The part must be properly connected according to its multi-rate properties. You will probably want to set the system rate in the *Data Flow Analysis* (sim) in order to obtain results compatible with the SystemVue 2007 simulation.

ℹ	Since SystemVue uses a different random number generator, your SystemVue 2007 simulations that have random signal and noise sources may not produce exactly identical results even when the random seed is fixed.

The APG DLL is used during the simulation run, so it must remain in the same location.

Continuing Development

You don't have to abandon your SystemVue 2007 design after importing it into SystemVue. As long as the number of inputs and outputs and their multi-rate properties remain the same, you can go back to modify the MetaSystem - change parameters and even add new tokens - and then simply re-generate the APG, overwriting your old DLL. You don't even have to close the SystemVue session - just make sure the simulation is not running. Your SystemVue design will continue working.

If you change the number of inputs or outputs or their multi-rate properties, you will need to re-import the APG DLL and modify your SystemVue design accordingly.

Using X-Parameters in SystemVue (RF Design Kit)

This section shows how X-Parameter data can be incorporated into SystemVue designs.

The X-parameter model is a generalized **circuit model** that includes **nonlinear** effects. The data for this model is contained in a Generalized MDIF file. A non-linear circuit simulation technique called **Harmonic Balance** is needed to make sense of X-parameter data.

X-Parameters are used in RF circuits to represent non-linear incident and reflected traveling waves.

Contents

Convergence Issues

The X-parameter model is a circuit level component. A non-linear circuit simulation technique called **Harmonic Balance** is needed to make sense of X-parameter data. Under high nonlinear conditions harmonic balance may be unable to converge to an accurate solution. In these cases, convergences parameters can be tweaked to optimize convergence for the given circuit problem.

By default when **XPARAMS** models are combined with system behavioral models in the same design each of the XPARAMS models will use the same generic default convergence criteria. This model provides no mechanism for the user to change the convergence criteria. When XPARAMS model(s) are placed in a Circuit_Link design the the entire design will all have common convergence criteria that can be controlled by the user.

Getting X-Parameters into the Workspace

X-parameters file data will automatically be imported and cached into memory when a simulation runs that contains X-parameter parts. All X-parameter data used in a workspace will remain in cached memory until the workspace is close or another workspace is opened.

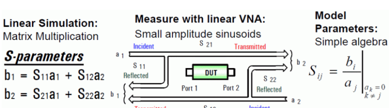
> **Note**
> Neither datasets nor any other type of workspace tree object is created during this automatic import process. X-parameter file data is cached to improve simulation performance.

For more information on the X-parameter file format see *X-parameter GMDIF Format* (users).

Theory of Operation

Traditional S-Parameters

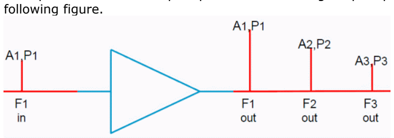
At high RF frequencies terminal voltages and currents are difficult to measure. Scattering parameters, or S-parameters are ratios of power flow amplitudes and phases in a circuit which are much easier to measure at these frequencies. However, S-parameters only characterize the linear behavior of RF devices.



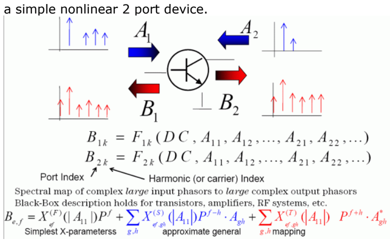
X-Parameter Basics

Unlike S-parameters, X-parameters characterize the linear and non-linear circuit behaviors of RF components in a more robust and complete manner. In effect, X-parameters are the mathematically correct super-set of S-parameters, applicable to both large-signal and small-signal conditions, for linear and nonlinear components. X-parameters are cascade-able just like S-parameters so higher levels of integration can be simulated or characterized.

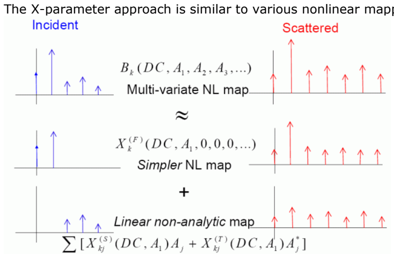
A simplified non-linear output spectrum from a single input spectrum is shown in the following figure.



The incident waves A1 and A2 and the resultant reflected B1 and B2 waves are shown for a simple nonlinear 2 port device.



The X-parameter approach is similar to various nonlinear mapping techniques as shown.



File Extraction Basics

X-parameter data can either be extracted by special network analyzers such as Agilent's

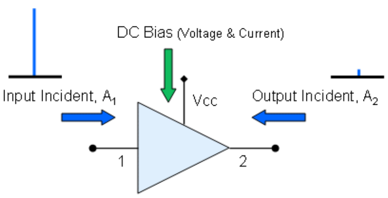
NVNA network analyzer or specialized simulation software such as Agilent's Advanced Design System (ADS). When an X-parameter file is extracted from a nonlinear device the user must supply the following extractions parameters and boundaries:

1. The number of characterization carriers (large signal).
2. The frequency of each carrier (*fund_k*).
3. The power level range of each carrier (*AN_p_n*).
4. The phase range of each carrier (*AP_p_n*).
5. The characteristic impedance.
6. DC voltage or current bias ranges (*VDC_p* & *IDC_p*).
7. Load characteristics that may be in the form of either reflection coefficients or impedance's (*GM_p_n, _GP_p_n*, etc).
8. User specified variables may also be used

Notation:

- **k** - fundamental frequency index
- **p** - port index
- **n** - harmonic index
- **m** - minus sign i.e. *_m2* = -2

Example of setup for 1 characterizing tone (Output Incident, A_2 is optional):



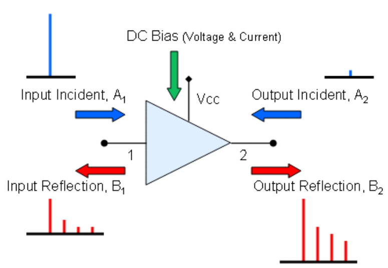
Extracted Data

Specialized hardware or simulation software extracts a text file containing the dependent data based on the independent input parameters listed in the prior section. The extracted output consists of several pieces of information for each input carrier. Every port is examined across a specified range of harmonics of the input carriers. Each piece of the contributing resultant output spectrum is characterized and saved in the extracted file.

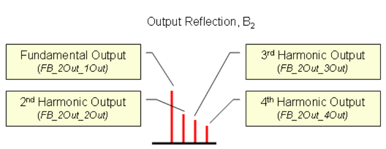
The extracted output consists of the following data:

1. Carrier reflected wave at the output (*FB_pOut_nOut*)
2. DC output current (*FI_pOut*)
3. DC output voltage (*FV_pOut*)
4. Small signal added output contribution due to a small signal input (S_pOut_nOut_pIn_nIn)
5. Small signal added output contribution due to phase-reversed small signal inputs (T_pOut_nOut_pIn_nIn)
6. DC current added output contribution due to small signal inputs (*XY_pOut_pIn_nIn*)
7. DC voltage added output contribution due to small signal inputs (*XZ_pOut_pIn_nIn*)

Example of extracted data from a single large signal characterizing tone:

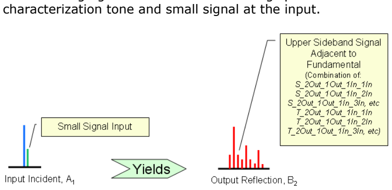


Examining the **Reflected B $_2$** spectrum for the 1 characterizing input tone we get:



To account for large and small signal effects a 'Quasi-Linear' system is created by internally generating a small signal at frequencies slightly different than the characterizing carrier frequencies. These small signals combined with the large characterizing signals produce new frequencies. By linear superposition the output frequencies and amplitudes can be determined for all small signal inputs in a real system.

The following figure illustrates the resulting spectrum from a single large signal characterization tone and small signal at the input.



For more information see *X-parameter Variables* (users).

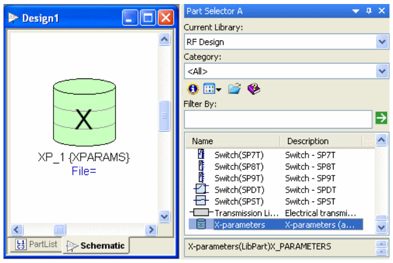
Using X-Parameters in a Design

To use an X-parameter file in a design follow these steps:

1. Place an X-Params Part
2. Browse to the X-Parameter File
3. Finish the Design
4. Add an Analysis

Place an X-Params Part

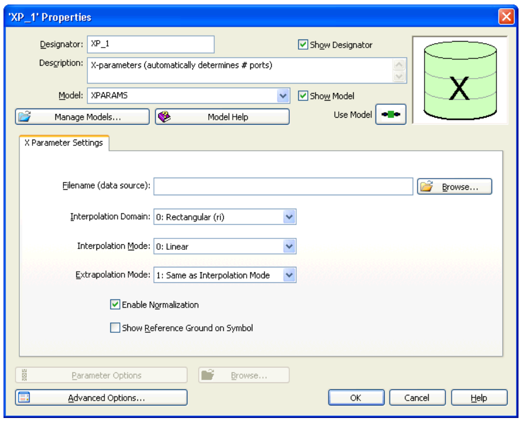
Select the **X-Params** part from the part selector located in the **RF Design** library.

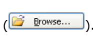


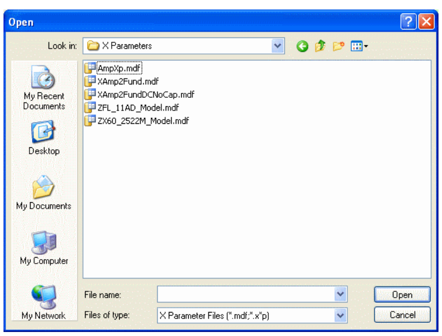
Note
When the X-Params model is placed the schematic symbol contains no pins. This is because the X-parameter file has not yet been selected (and of course, has not been read); the number of ports cannot be determined until the file is actually read.

Browse to the X-Parameter File

Double click the X-Params part to bring up the part properties.

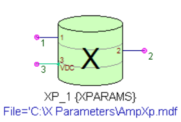


Click the Browse ().

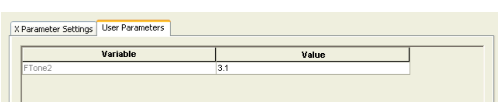


Select the desired X-parameter file.

At this time the number of ports is resolved and the schematic symbols changes appropriately because a specified X-parameter file has been selected.

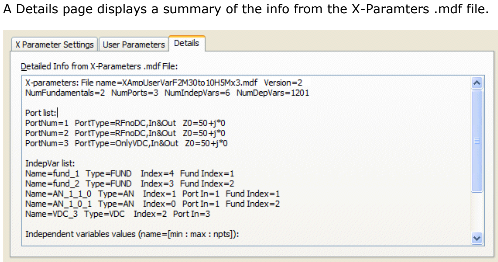


Also, one or two additional (optional) tab pages may appear: A User Parameters tab may appear, if the X-parameter file has any User Variables defined.



These parameters are defined by the file. You may NOT add, delete, or rename the User Variables, but you can change their values to any floating point number. (Equations are not permitted for values.)

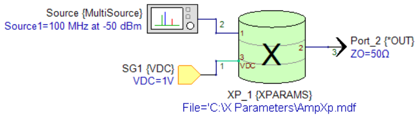
A Details page displays a summary of the info from the X-Paramters .mdf file.



For more information on the X-Params model properties see *X-Parameter Part* (rfdesign).

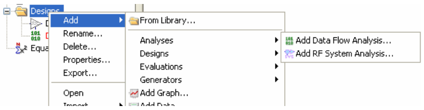
Finish the Design

Place the desired components to finish the design. (In this particular example a Multisource, Output Port, and Signal Ground parts are used)



Add an Analysis

Add the desired analysis.



Run the analysis and plots the results.

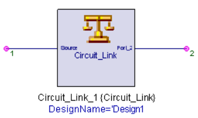
Using DC Bias Voltage

X-parameters can be characterized with various DC bias voltages and can even support multiple DC bias ports.

Caution
If the X-parameter file has been characterized with a single DC voltage the internal interpolation and extrapolation algorithms can only use this **single** bias point so all interpolated or extrapolated DC bias voltages will all be at the same DC bias voltage. Consequently, specifying a DC bias voltage on the part becomes irrelevant.

Using X-Parameters in the Circuit Link

The **Circuit Link** component is used as a bridge between circuit and system level components.



This bridge points to a design and contains parameters most often needed to control circuit level convergence criteria. A nonlinear circuit simulation technique called harmonic balance uses this criteria to simulate the linear and nonlinear characteristics of the circuit. These results are passed to the Spectrasys for spectral creation and path measurement calculations.

For more information see *Circuit_Link* (rfdesign)

Caution
The accuracy of cascaded circuit components will be increased when all circuit level components are combined in a single *Circuit_Link* (rfdesign) component.

Using X-Parameters in the RF Link (RF Design Kit)

The **RF Link** characterizes the system design with a single frequency. This characterization takes place across a frequency range extracted from the DataFlow analysis, unless this is overridden by the user in the RF Link component. The RF Link power characterization range is identical to the power range specified when the the X-parameter file was initially extracted. The frequency at which the power characterization takes places is in the center of the frequency characterization range.

This characterization method has current limitations on the types of X-parameter files that may be used in RF_Link:

1. **No Frequency translation**. Only X-parameter files that have the same input and output frequencies (amplifiers, attenuators, filters, etc.) are allowed.
2. **Only 2-port Circuit_Link / X-parameter files are supported**. If a Circuit_Link component or X-parameter file has more than 2 ports the current characterization methods does not have any information about other port signals or DC port states. Consequently, characterization data may be inaccurate.

Note
Since the link uses a one tone characterization technique then the first tone in the X-parameter file should also be swept to include frequency response. If a two or more tone X-parameter file is used and the first tone is NOT swept then the frequency response will be constant.

Caution
The RF Link does power compression characterization of the X-parameter device. **Terminal 0** is always used for the **input** and **terminal 1** for the **output**. Caution must be used when generating the X-parameter file so that the input is terminal 0 and output terminal 1.

Using X-Parameters in Spectrasys

The X-Params model predicts the circuit level output currents and voltages given specific characterization characteristics contained in the X-parameter file. A common nonlinear simulation technique called **Harmonic Balance** is used to extract the necessary information needed by the system simulator called Spectrasys. Spectrasys is a nonlinear behavioral simulator and the simulation approach is drastically different than that used for nonlinear circuits.

RF system simulation is used to determine optimum RF architecture as well as requirements for each of the behavioral blocks or sub-systems in a system that has a common characteristic impedance. Circuit simulations can be oblivious to characteristic impedance's and users are generally more interested in circuit input and output characteristics rather than cascaded parameters are some internal intermediate nodes.

Note
Highest circuit simulation accuracy will be achieved when all circuit level components such as X-parameters are placed together in a single **Circuit Link** component. Complex circuit level interactions between cascaded circuit components may be missed in during the system simulation.

Validation Limits

1. Spectrasys simulation using single X-parameters part with single tone or 2-tone stimulus has been compared with equivalent simulation in ADS, all results are consistent.
2. Spectrasys simulation using cascaded X-parameters parts with single tone or 2-tone stimulus has been compared with equivalent simulation in ADS, results are consistent with reasonable (negligible) difference (e.g. less than a few tenths of a dB at fundamental frequency and can be slightly higher for mixing terms < -50dBm) due to the difference in underlying computational algorithms (e.g convergence criteria).

Performance Limits

If simulation speed becomes an issue (most likely due to convergence), use Circuit_Link with *X-Parameters Part* (rfdesign) part to control the convergence criteria directly.

Operational Limits

> **Caution**
> Currently, X-parameter models are not allowed in the LO chain for **RF LINK** simulations only.

Noise

> **Note**
> Currently, the X-parameter parts do not support self generated device noise. However, any external noise appearing at the X-parameter ports will be amplified by the small-signal gain specified in the X-parameter file.

Frequency and Power Limits

X-parameter files are extracted across a user specified power range with a fixed number of input tones at user specified frequencies. During a simulation frequency and power values will be interpolated if the simulation frequencies and power levels reside within the characterization limits otherwise the values will be extrapolated.

> **Caution**
> If the characterizing tones are not swept in frequency or power there will be noting to interpolate or extrapolate since all frequency and power levels will appear to be constant.

Tone Characterization and Mapping

Along with frequency and power level characterization a non-linear circuit is characterized by a fixed number of input tones (carriers) specified by the user. Furthermore, these tones can be swept or fixed in frequency and power level. During a simulation three simulation scenarios exist with regard to the number of tones used in the simulation versus the number of tones the X-parameter file was characterized with. They are:

1. Number of Simulation Tones **=** Number of X-parameter Characterization Tones
2. Number of Simulation Tones **<** Number of X-parameter Characterization Tones
3. Number of Simulation Tones **>** Number of X-parameter Characterization Tones

> **Note**
> Highest accuracy will only be achieved when the X-parameters are extracted with the exact number of carriers, frequencies, and power levels of interest.

X-parameters are simulated using a **large-signal-small-signal analysis** technique. In this technique a certain number of tones are designated as large signal all other input signals are considered small signal. When the large and small signal analysis techniques are combined distortion (intermod) products can be determined at all distortion frequencies.

During an X-parameter simulation all input carriers are sorted by power level. The largest input signal maps to the 1st X-parameter tone and the 2nd largest input signal maps to the 2nd X-parameter tone, etc. until all the large signal tones have been mapped. For example, if an X-parameter file was characterized with two tones, the first one fixed in frequency and power, and the second swept in power and frequency then during the simulation the largest power input tone would map to the fixed X-parameter tone and the next input carrier would map to the swept characterizing tone.

If the number of simulation tones equals the number of X-parameter characterization tones then each input tone is considered a large signal tone. If the number of simulation tones is less than the number of X-parameter characterization tones then the extra X-parameter characterization tones are ignored. If the number of simulation tones is greater than the number of X-parameter characterization tones then all the unmapped tones become small signal input tones.

> **Caution**
> If the X-parameter file was only characterized with one tone and two tones are being used in the simulation the resulting simulation will be a one tone large signal analysis with a single small signal not the traditional two tone analysis.

For more information on **large-signal-small-signal analysis** see Mass, Stephen A, *Nonlinear Microwave Circuits.* Norwood, MA: Artech House, 1988, Chapter 3.

Appendix A - Keystroke Commands

- *General Keystroke Commands* (users)
- *Graph Keystroke Commands* (users)
- *LiveReport Keystroke Commands* (users)
- *Schematic Keystroke Commands* (users)

The availability of keystroke commands depends on the type of active window (Graph, Schematic, etc.).

General Keystroke Commands

- **Space** – Place another copy of the most recently placed item (schematics and layouts)
- **Escape** – Cancel current mode
- **Delete** – Delete current selection
- **Ctrl+A** – Select all
- **Ctrl+C** – Copy
- **Ctrl+D** – Duplicate
- **Ctrl+N** – File new
- **Ctrl+Shift+N** – Select none
- **Ctrl+O** – File open
- **Ctrl+P** – Print
- **Ctrl+S** – Save
- **Ctrl+V** – Paste
- **X** – Zoom – use the zoom tool (zoom to mouse rectangle)
- **Ctrl+X** – Cut
- **Ctrl+Y** – Redo
- **Ctrl+Z** – Undo
- **Z** – Zoom to fit all objects (Maximize)
- **Shift+Z** – Zoom to fit with extra margin
- **Ctrl+Shift+Z** – Redo
- **+** – Zoom in
- **–** – Zoom out
- **Ctrl+End** – Show entire page (maximize)
- **Ctrl+Home** – Zoom to fit
- **Ctrl+PageUp** – Zoom in
- **Ctrl+PageDown** – Zoom out
- **LeftArrow**, **RightArrow**, **UpArrow**, **DownArrow** – Move the current selection in the direction indicated (use the Enter key to drop parts in schematic after moving with the arrow keys)
- **Ctrl+LeftArrow**, **Ctrl+RightArrow**, **Ctrl+UpArrow**, **Ctrl+DownArrow** – Pan (scroll) the view (when nothing is selected)
- **F3** – Rotate item clockwise
- **Shift+F3** – Rotate item counterclockwise
- **Ctrl+F3** – Reset rotation angle to 0
- **F5** – Does an Action / Run All Out-of-Date Analyses and Sweeps (calculates simulations/sweeps)
- **Shift+F5** – Run all optimizations
- **F6** – Mirror an item
- **Ctrl+F6** – Reset mirror state to unmirrored
- **Alt+F7** – Print/export entire screen
- **F7** – Hide/Show docker windows (tree and tune windows)
- **F8** – Fit Windows to Frame - resize the windows to fit the non-docker area
- **Ctrl+F8** – Next editor
- **Alt+F8** – Print/export active window

Graph Keystroke Commands

- **C** – Checkpoint – Create a graph checkpoint or remove existing checkpoints
- **F** – Favorite – save a graph axis favorite
- **B** – Back – use a graph axis favorite
- **V** – Vertex – Hide / Show vertex symbols
- **R** – Right – Show markers on right / floating
- **M** – Mark – mark all traces with markers
- **L** – Legend – hide/show the legend
- **P** – Pan – use the pan (scrolling) tool
- **X** – Zoom – use the zoom tool (zoom to mouse rectangle)
- **Z** – Zoom to fit – Maximize the view
- **Tab** – Select the next marker.
- **Shift+Tab** – Select the previous marker.
- **Enter** – Bring up the Marker Properties window. If no marker is selected, it brings up the Graph Properties instead.
- **Delete** – Delete the currently selected marker.
- **Shift+Delete** – Delete all markers (you are asked to confirm the deletion before deleting the markers).
- **Arrow Keys** – The up, down, left, and right arrow keys have several functions, based on the currently selected marker's style.
    - **Standard Marker** – Move the reference frequency left or right on the graph.
    - **Peak Marker** – Move to the next peak (if any).
    - **Valley Marker** – Move the marker to the next valley (if any).
    - **Bandwidth Marker** – Move the relative markers to increase or decrease the bandwidth. This changes the delta values of the child relative markers, so each arrow key action does not always move the marker by a single data point.
    - **Delta Marker** – Increase or decrease the relative delta. This changes the dB Down value of the marker, so each arrow key action does not always move the marker by a single data point.
- **Ctrl+Arrow Keys** – Pan (scroll) the chart up, down, left, or right.
- **Shift+Arrow Keys** – Move the marker up or down to the next trace on the graph (if any).
- **Ctrl+Shift+S** – Change the current marker's style to Standard.
- **Ctrl+Shift+P** – Change the current marker's style to Peak.
- **Ctrl+Shift+V** – Change the current marker's style to Valley.
- **Ctrl+Shift+B** – Change the current marker's style to Bandwidth.
- **Ctrl+Shift+L** – Change the current marker's style to Delta Left.
- **Ctrl+Shift+R** – Change the current marker's style to Delta Right.

LiveReport Keystroke Commands

- **A** – All Zoom - zoom to page
- **P** – Pan - use the pan (scrolling) tool
- **X** – Zoom - use the zoom tool (zoom to mouse rectangle)
- **W** – Zoom to Width
- **Z** – Zoom to fit - Maximize the view
- **Tab** – switch to next window
- **Shift+Tab** – switch to previous window
- **1, 2, 3, 4, 5, ...** – switch to nth window (zooms to fit specified window)

Schematic Keystroke Commands

- **Enter** – Bring up part properties or place parts moved using the arrow keys
- **A** – Places an adder (Add)
- **B** – Bits (Source: Bits)
- **C** – Const (Source: Const)
- **D** – Delay
- **Shift+D** – DownSample
- **G** – Gain
- **I** – DataPort (input)
- **M** – MathLang
- **O** – DataPort (output)
- **P** – Use the Pan (scrolling) tool
- **R** – Ramp (source)
- **S** – Sink
- **Shift+S** – SineGen
- **U** – Upsample
- **W** – 90 degree WIRES (Shift+W for any angle wires)
- **Shift+W** – Angled WIRE
- **X** – Zoom - use the zoom tool (zoom to mouse rectangle)
- **Z** – Zoom to show all parts (zoom to fit)
- **Shift+Z** – Zoom to show all parts (with extra margin)
- ***** – Mpy (multiply)

- **F4** – Rotate the text origin of part parameters

If the schematic has RF (Spectrasys) parts on it, the following key / part associations are used.

- **Enter** – Bring up part properties or place parts moved using the arrow keys
- **A** – Places an ammeter (CURRENT_PROBE)
- **B** – BLOCK (two-port)
- **C** – CAPQ (capacitor with Q)
- **Shift+C** – CAPACITOR (ideal)
- **G** – GROUND
- **I** – INPUT Port
- **L** – INDQ (inductor with Q)
- **Shift+L** – INDUCTOR (ideal)
- **O** – OUTPUT port
- **P** – Use the Pan (scrolling) tool
- **Q** – SQUARE_BLOCK (attached to a design)
- **R** – RESISTOR
- **S** – SIGNAL_GROUND
- **V** – Voltage TEST_POINT
- **W** – 90 degree WIRES (Shift+W for any angle wires)
- **Shift+W** – Angled WIRE
- **X** – Zoom - use the zoom tool (zoom to mouse rectangle)
- **Z** – Zoom to show all parts (zoom to fit)
- **Shift+Z** – Zoom to show all parts (with extra margin)
- **F4** – Rotate the text origin of part parameters
- **1**, **2**, **3**, ..., **0** – Place 1-port, 2-port, ..., 10-port

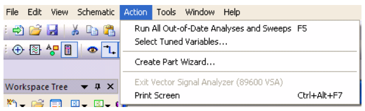
Appendix B - Menus

- *Action Menu* (users)
- *Edit Menu* (users)
- *Equations Menu* (users)
- *File Menu* (users)
- *Graph Menu* (users)
- *Help Menu* (users)
- *LiveReport Menu* (users)
- *Notes Menu* (users)
- *PartList Menu* (users)
- *Schematic Menu* (users)
- *Scripts Menu* (users)
- *Tools Menu* (users)
- *View Menu* (users)
- *Window Menu* (users)

Action Menu

Use this menu to calculate variables or to access the Create Part, Design, or Source wizards.

To open: Click the Action button on the menu.

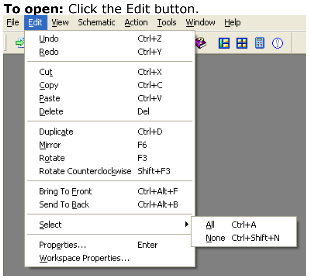


1. **Calculate** – Calculate the out-of-date simulations.
2. **Calculate All Optimizations** – Run all the optimizations.
3. **Select Tuned Variables** – Make any parameter from a master list tunable.
4. **Create Part Wizard** – Run the part creation wizard. Use this to create a new part based on existing parts or from scratch by defining the model and symbol for the part.
5. **Print Screen** – print the current screen.

Edit Menu

Use this menu to perform basic editing functions, such as undo, redo, cut, paste, copy, and delete.

To open: Click the Edit button.

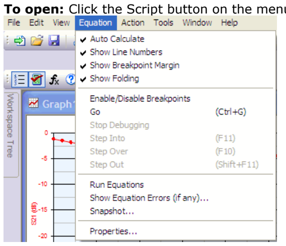


1. **Undo** – Reverse previous editing. Multi-level undo is available in a schematic or layout.
2. **Redo –** Put back changes that were previously reversed with Undo.
3. **Cut –** Copy the selected object and delete it.
4. **Copy –** Copy the selected object. The selection is not deleted.
5. **Paste** – Paste the last copied object into the current schematic, layout, text, etc.
6. **Delete –** Delete the selected object.
7. **Duplicate** – Duplicate the selected object. This is equivalent to a copy-and-paste sequence.
8. **Mirror** – Flip the selected object about its horizontal or vertical axis. Mirror is not available for layouts, because it yields backward parts.
9. **Rotate** – Rotate the selected object by the Part Constrain angle specified in the Global Schematic Options window.
10. **Bring To Front** – Moves the selected item(s) in front of the other items in the window.
11. **Send To Back –** Moves the selected item(s) behind the others.
12. **Rotate Counterclockwise** – Rotate the selected object counterclockwise.
13. **Select** – Display a submenu allowing easy access to commonly used objects
14. **All** – Select all objects in the schematic or layout.
15. **None** – Turn off all selected objects.
16. **Properties** – Open properties for active object.
17. **Workspace Properties** – Open workspace properties.

Equations Menu

Use this menu to access equations commands.

To open: Click the Script button on the menu.



1. **Active** – When checked, this equation is available for use.
2. **Auto Calculate** – When checked, these equations will recalculate while typing

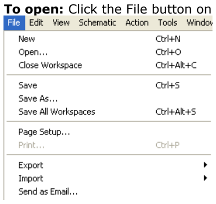
   ⚠ Caution: be careful not to write infinite loops if this option is checked

3. **Show Line Numbers** – Shows / hides line numbers in the equations window.
4. **Show Folding** – Shows / hides the folding bar in the equations window (next to line numbers). When enabled, the folding bar can be used to expand/contract blocks of code, such as if / then / else sections.
5. **Equation Wizard** – Runs the Equation Wizard.
6. **Run Equations** – Executes the equation block.
7. **Show Equation Errors** – Helps diagnose equation errors.
8. **Snapshot** – Create a dataset with static variables that capture the current state of the equation block. Use it save reference variables, such as when the equation block is dependent on an analysis that gets re-run and you want to keep around old results in the workspace.
9. **Properties** – Shows the Equation's Properties dialog box

File Menu

Use this menu to open, close, save, or print designs. You can also import or export files, and exit.

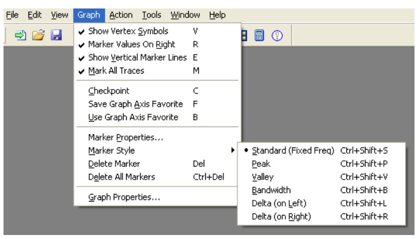
To open: Click the File button on the menu.



1. **New –** Close the current workspace and open a new workspace. If you select the Allow Multiple Open Workspaces option on the General Global Options page, the current workspace remains open.
2. **Open –** Opens a new workspace.
3. **Close Workspace** – Close the current workspace.
4. **Save –** Save the current workspace. If the current file has not been previously saved, you will be prompted for a file name.
5. **Save As** – Save the current workspace into a new file.
6. **Save All Workspaces –** Save all loaded workspaces.
7. **Page Setup** – Select printer and settings.
8. **Print –** Print the active window.
9. **Export** – Display a submenu allowing access to all of the Export options.
   - **Bitmap (Active Window)** – Export the active window.
   - **Bitmap (Entire Screen)** – Export the entire screen, including any applications outside the window.
   - **XML File** – Export the published properties to an XML file.
10. **Import** – Display a submenu allowing access to all of the Import commands.
    - **M-File** – Import an M-file.
    - **Directory of M-Files** – Import all M-files in a directory
    - **S-Data file** - Import an S Parameter file in Touchstone format.
    - **SPICE File** – Import a SPICE file.
    - **XML** – Import an XML file.
    - **CITI File** – Import a Common Instrumentation Transfer and Interchange (CITI) file.
11. **Send as Email** – Send the current workspace as an email attachment using your email program.

Graph Menu

Use this menu to specify various graph settings. **To open:** Click the Graph button on the menu. (This menu appears only when a graph window is active.)



1. **Show Vertex Symbols** – Show or hide the vertex symbols on the trace.
2. **Marker Values On Right --** Place marker values on the right of the graph.
3. **Show Vertical Marker Lines** – Show or hide the vertical marker lines.
4. **Mark All Traces --** Place markers on all traces.
5. **Checkpoint --** Remove all current checkpoint traces if there are any. Create one if there are none.
6. **Marker Properties** – Open the Marker Properties window.
7. **Marker Style** – Display a submenu allowing easy access to commonly used marker styles.
8. **Standard (Fixed Frequency)** – Place a maker on the graph at the sport where you clicked.
9. **Peak** – Place a marker at the highest point on the trace.
10. **Valley** – Place a marker at the lowest point on the trace.
11. **Bandwidth** – Placer a marker on the trace to indicate bandwidth.
12. **Delta (On Left)** – Place a marker left of the trace to indicate the relative offset specified in the Marker Properties window.
13. **Delta (On Right)** – Place a marker right of the trace to indicate the relative offset specified in the Marker Properties window.
14. **Delete Marker** – Delete the currently selected marker.
15. **Delete All Markers** – Delete all the markers on the current graph; it prompts yes/no before actually deleting the markers.
16. **Graph Properties** – Open the Graph Properties window.

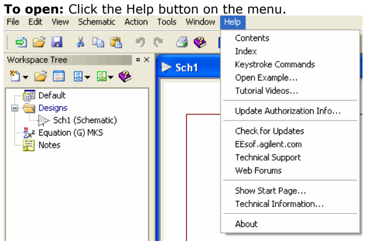
See Also

- *Graphs* (users)
- *Types of Graphs* (users)
- *Graph Properties* (users)
- *Graph Toolbar* (users)
- *Using Markers on Graphs* (users)
- *Tables* (users)

Help Menu

Use this menu to check for the latest update, get quick access to the Agilent Web site, or get help.

To open: Click the Help button on the menu.



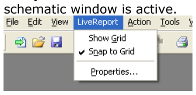
1. **Contents** – Open the Help contents.
2. **Index** – Open the Help index.
3. **Keystroke Commands** – Open a Help topic containing information about all of the keystroke commands.
4. **Open Example** – Open an example workspace.
5. **Tutorial Videos** – Select and watch a collection of short, helpful videos.
6. **Update Authorization Information** – Open a page where you can start the authorization process.
7. **Check for Updates** – Open a Web page to check for updates.
8. **Agilent.com** – Open the Agilent Web site.
9. **Technical Support** – Open the technical support Web page.
10. **Web Forums** – Open the Web page to access one of the forums.

11. **Show Start Page** – Open the Start page.
12. **About** – Open a page with information about the program.

LiveReport Menu

Use this menu to set LiveReport options. (A LiveReport is a *living* notebook page that collects live views of schematics, graphs, equations, notes, and tables into a single page.)

To open: Click the Schematic button on the menu. This menu appears only when a schematic window is active.

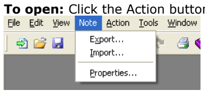


1. **Show Grid** – Show or hide the background grid.
2. **Snap to Grid** – Toggles (enables / disables) mouse cursor snap-to-grid (constrains mouse coordinates to the grid).
3. **Properties** – Shows the LiveReport Properties dialog box, which allows you to specify settings such as Page Width and Height, Paper Orientation, Margins, Headers, and Footers.

Notes Menu

Use this menu to access Note commands.

To open: Click the Action button on the menu.



1. **Export** – Export the Note's text.
2. **Import** – Import text into the note.
3. **Properties** – Shows the Note's Properties dialog box

> In order for the **Note** menu to reveal, the Notes page must be the current selected window (either open or minimized) in the SystemVue workspace area.

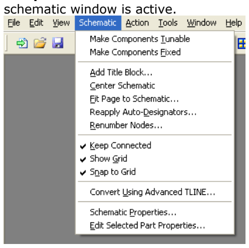
PartList Menu

The PartList has a single item:
Properties – Open the Properties window.

Schematic Menu

Use this menu to set component and schematic options.

To open: Click the Schematic button on the menu. This menu appears only when a schematic window is active.

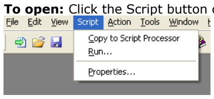


1. **Make Components Tunable** – Force selected components to be tunable or optimizable by adding question marks (?) to the first value of each component. This only adds question marks to part values with a numerical value. If a variable is used for a particular value, it is not made tunable.
2. **Make Components Fixed** – Force selected components to be non-tunable by removing any question marks that were added to the first value of each component. This only removes question marks on part values with a numerical value.
3. **Add Title Block** – Adds a schematic title block to the page, so that the schematic can be documented.
4. **Center Schematic** – Center the schematic on the page.
5. **Fit Page to Schematic** – Resize the page to fit all the parts within it. Note that you can also change the standard part length in a schematic to have parts shrink to fit a specific page size.
6. **Reapply Auto-Designators** – Reassign standardized designators to selected components. A designator is a part name like R1 or C3. The Auto-Designator feature builds component names by using the appropriate designator prefix (like R for a resistor or C for a capacitor) and appending a unique sequence number to the end. When you use this command, the designators are applied in geometric order, from left to right.
7. **Renumber Nodes** – Renumber all nodes in the schematic, regardless of any selection. When you use this commend, the nodes are numbered in geometric order, from left to right. Nodes that connect to a port are set to match the port number (if that option is enabled). This is primarily useful before exporting a SPICE file.
8. **Bring to Front** – Move the selected objects to the front.
9. **Send to Back** – Move the selected objects to the back.
10. **Keep Connected** – Allow wires to remain connected to components as they are moved. The ALT key temporarily toggles this function as long as the key is held down.
11. **Show Grid** – Show or hide the schematic grid.
12. **Snap to Grid** – Toggles (enables / disables) mouse cursor snap-to-grid (constrains mouse coordinates to the grid).
13. **Convert Using Advanced TLine** – Convert all electrical transmission line parts to physical transmission line parts using Advanced TLine (for example, microstrip, stripline, coplanar, or coax). This allows discontinuities to be added and automatically compensated for. Also, substrates can be converted from one to another.
14. **Schematic Properties** – Shows the Schematic Properties dialog box, which allows you to specify settings such as Page Width and Height, Title, Company Name, and Company Address.
15. **Edit Selected Part Properties** – Shows the Part Properties dialog box, which allows you to specify parameters and settings for the selected part.

Scripts Menu

Use this menu to access scripting commands.

To open: Click the Script button on the menu.

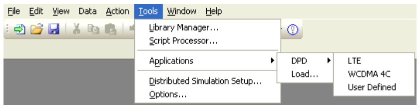


1. **Copy to Script Processor** – Copies the script to the Script Processor window.
2. **Run** – Executes the script.
3. **Properties** – Shows the Script's Properties dialog box

> The script menu shows only when a script page is present.

Tools Menu

Use this menu to access to some common design tools or change the global options.

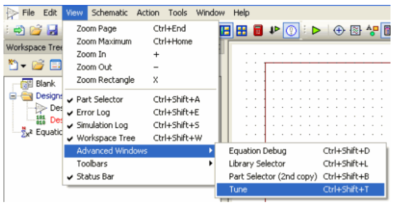


1. **Library Manager** – Open the Library Manager window, which controls which libraries are initially loaded.
2. **Script Processor** – Open the Script Processor window, to run VBScript or JScript commands.
3. **Applications**
    1. **DPD** – Run LTE, WCDMA 4C, or User Defined
    2. **Load** – Load an assembly
4. **Distributed Simulation Setup** – Open the Distributed Simulation Setup window, to set Host Name, User Name, and Public Key for distributed simulations.
5. **Options** – Open the Global Options window, which controls number formatting, graph and schematic settings, unit defaults, etc.

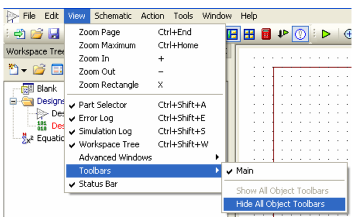
View Menu

Use this menu to adjust the size of your window. This menu can also be use to show or hide docking windows or toolbars.

To open: Click the View button on the menu.



- **Zoom In** – Zoom in on the center of the window.
- **Zoom Out** – Zoom out from the center of the window.
- **Zoom Page** – Zoom to fit the page.
- **Zoom Maximum** – Zoom to fit all objects or traces.
- **Zoom Rectangle** – Allow you to draw a rectangle to zoom in on.
- **Part Selector** – Show or hide the Part Selector.
- **Error Log** – Show or hide the Error Log.
- **Simulation Log** – Show or hide the Simulation Log.
- **Workspace Tree** – Show or hide the Workspace tree.
- **Advance Windows** – Show a secondary list of docking windows.
    - **Equation Debug** – Show or hide the Equation Debug window.
    - **Library Selector** – Show or hide the Library(Design) Selector.
    - **Part Selector (2nd copy)** – Show or hide a second copy of the Part Selector.
    - **Tune** – Show or hide the Tune window, which lists and controls tune variables.

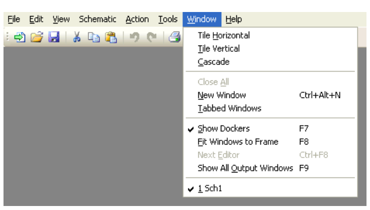


- **Toolbars** – Choose how toolbars are shown.
    - **Main** – Show or hide the Main toolbar.
    - **Show All Object Toolbars** – Show toolbars for the active object.
    - **Hide All Object Toolbars** – Hide toolbars for the active object.
- **Status Bar** – Show or hide the status bar at the bottom of the main window.

Window Menu

Use this menu to organize or open a window. You can also use this menu to close all open windows at the same time.

To open: Click the Window button on the menu.



1. **Tile Horizontal** – Tile open windows above each other.
2. **Tile Vertical** – Tile open windows beside each other.
3. **Cascade** – Arrange open windows in an overlapping style.
4. **Close All** – Close all open windows.
5. **New Window** – Open a new design window.
6. **Tabbed Windows** – Switches between tabbed and overlapping document window styles.
7. **Show Dockers** – Show / hide vertical dockers (Tune, Workspace Tree, Part Selector, etc.).
8. **Fit Windows to Frame** – Resizes the open windows to fit the non-docker area.
9. **Next Editor** – Toggle between editor windows (schematics, layouts, equation editors).
10. **Show All Output Windows** – Open all output windows (graphs, tables, variable viewers).
11. **Numbered Window List** – A pick-list of all open document windows. Select one to make it active (current).

Appendix C - Toolbars

- *Annotation Toolbar* (users)
- *Dataset Toolbar* (users)
- *Equation Toolbar* (users)
- *Graph Toolbar* (users)
- *LiveReport Toolbar* (users)
- *Main Toolbar* (users)
- *Notes Toolbar* (users)
- *Schematic Toolbar* (users)
- *Script Toolbar* (users)
- *Spectrasys Toolbar* (users)
- *Table Toolbar* (users)

Annotation Toolbar

Use this toolbar to add basic drawing objects, such as lines, circles, or arrows, to a design or to modify the selected annotations by changing the color, dashed line style, etc.

To open: Click the Annotation button ( ) from any design window toolbar, e.g. schematic window toolbar.

1. **Select** – Select an object.
2. **Rectangle** – Draw a square or rectangle.
3. **Ellipse** – Draw a circle or ellipse.
4. **Polygon** – Draws a filled polygon or unfilled polyline.
5. **Arrow** – Draw a line or arrow. Change the arrow style by selecting a line and picking an arrow type from Arrows button menu.
6. **Arc** – Draw an arc.
7. **Picture** – Insert a picture. Use this annotation to add a company logo to a graph, for example. Double-click the new object and select a JPG, GIF, or BMP image file to be displayed. To allow all users to see the image, the bitmap file should reside on a network server.
8. **Text** – Place text. Text has a number of settings. Double-click a text annotation to set the horizontal and vertical justification (text alignment). The name of the text item can be changed and shown on-screen, which simplifies building a schematic title block.
9. **Text Balloon** – Draw a text balloon. This annotation has a "tail" which can be anchored to a data point on a graph, to the page, or not anchored (using the right-button menu).
10. **Button** – Draw a user button (widget). This annotation can be "clicked" to run a custom script, which is specified by double-clicking the outer EDGE of the button control. The middle of the button runs the script.
11. **Slider** – Draw a slider control (widget). This annotation is linked to a tunable parameter and functions much like the Tuning Window.
12. **Fill Color** – Set the fill color. Use the 3 color buttons to change the colors of the selected annotations. New annotations will be created using the current colors. The bottom-right color swatch (with a diagonal slash) is transparent, which specifies an unfilled object.
13. **Line Color** – Set the line color. The bottom-right color swatch (with a diagonal slash) is transparent, which specifies a object with no outline.
14. **Text Color** – Set the text color.
15. **Line Thickness** – Set the width of borders and lines.
16. **Line Style** – Set the drawing style of borders and lines (dash pattern, etc.).
17. **Arrows** – Set the arrow style of lines.
18. **Properties** – Display the properties window for the selected part.

Dataset Toolbar

Use this toolbar to interact with the active *Dataset* (users) and adjust its settings.

1. **Properties** - Brings up the Dataset Properties dialog.
2. **Save** - Export/save the dataset to a file.

Equations Toolbar

Use this toolbar to change the Equation window display options and debug your equations. This toolbar automatically displays when you have an Equation window active.

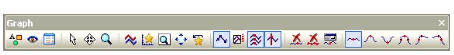
The icons are:
Show or hide the Line Numbers margin
Turn autocalculate on/off

Display Errors for this set of equations
Go - run the equations, or continue on from a breakpoint (F5 or Ctrl_G)
Stop - stop debugging (abort execution). This button is only enabled while breakpointed.
Step Into - step inside a function and break at the first line of execution in the function (F11). This button is only enabled while breakpointed.
Step Over - execute statements on the current line (F10). This button is only enabled while breakpointed.
Step Out - run until the current function ends, then break at the next line (the caller) (Shift_F11). If there is no function call at the current line, or the equation processor cannot step into the function, then all statements on the current line are simply executed. This button is only enabled while breakpointed.
Add or Remove a breakpoint from the current line (F9 or Ctrl_B)
Toggle all existing breakpoints to either the "enabled" or "disabled" state

Graph Toolbar

Use the toolbar for Graph functions.

To open: Open a graph window.



1. **Annotation** – Display the Annotation Toolbar toolbar.
2. **Eye** – Hide/Show graph traces in a pulldown menu
3. **Graph Properties** – Display the Graph Properties window.
4. **Select** – Select an object.
5. **Pan** – use the Pan tool to pan the graph (left-right for rectangular graphs, free for polar and smith charts).
6. **Zoom** – zoom in on a selected part of the graph.
7. **Checkpoint** – Add a checkpoint if there is none. Remove all current checkpoint traces if there are any
8. **Add Axis Favorite** – Save the current axis settings into the Axis Favorite list.
9. **Zoom to Page** – Zoom the graph data attractively to fit the page.
10. **Maximize** – Zoom the graph data exactly to fit the page..
11. **Use Axis Favorite** – Set the axis settings to the last favorite in the list. Click again to cycle through the axis favorites.
12. **Toggle Vertex Symbols** – Show or hide trace vertex symbols (large dots on traces).
13. **Marker Values On Right** – Place marker text in right margin of graph, or inline in graph.
14. **Mark All Traces** – Mark all traces on the graph.
15. **Toggle Vertical Marker Lines** – Show or hide dashed vertical marker lines at every

marker position.
16. **Delete Marker** – Delete the selected marker.
17. **Delete all Markers** – Delete all markers on the current graph.
18. **Marker Properties** – Display the Marker Properties window.
19. **Standard Marker** – drop a standard marker or convert a selected marker to standard.
20. **Peak Marker** – Change marker style to Peak.
21. **Valley Marker** – Change marker style to Valley.
22. **Bandwidth Marker** – Change marker style to Bandwidth and insert two Delta markers.
23. **Delta Marker (On Right)** – Place a new Delta marker on the left side of the selected marker.
24. **Delta Marker (On Left)** – Place a new Delta marker on the right side of the selected marker.

See Also

- *Graphs* (users)
- *Types of Graphs* (users)
- *Using Markers on Graphs* (users)
- *Graph Menu* (users)
- *Graph Properties* (users)

LiveReport Toolbar

Use this toolbar to change the LiveReport and adjust its settings. The LiveReport toolbar automatically displays when you have a LiveReport active.

1. **Annotation** – Show/Hide the Annotation Toolbar.
2. **Arrange** – Brings up the Arrange Views dialog box, which repositions all the sub-objects.
3. **Eye** – Use this pull down menu to turn on/off text displays such as Titles, Headers, Footers, etc. on the LiveReport.
4. **Grid Snap** – enable/disable the grid snap
5. **Select** – Use the select tool to select views or annotations..
6. **Pan** – Use the pan (scrolling) tool to pan the schematic around (press the tool button and drag the LiveReport).
7. **Zoom** – Use the zoom tool to zoom into a rectangular region of the LiveReport (press the tool button and drag a rectangle).
8. **Zoom to Page** – Zoom to fit the page.
9. **Zoom to Fit Selection** – Zoom to fit the currently selected objects.
10. **Zoom to Fit All** (Maximize) – Zoom to fit all objects.
11. **Properties** – Opens the LiveReport properties dialog box.

Main Toolbar

Use this toolbar for global functions, like File Save, Print, and Undo.

To open: Click View on the menu and select Main from the Tools menu.

1. **Start Page** – Create a new workspace.
2. **Open** – Open an existing document.
3. **Save** – Save the active document.
4. **Cut** – Cut the selection to the clipboard.
5. **Copy** – Copy the selection to the clipboard.
6. **Paste** – Paste the contents of the clipboard.
7. **Undo** – Undo the last action. Available only for schematics and layouts.
8. **Redo** – Redo the previously undone action. Available only for schematics and layouts.
9. **Print** – Print the active window.
10. **Help** – Open the Help file.
11. **Docker View Menu** – Drop down menu to allow dockers to be toggled hidden or shown.
12. **Hide/Show Dockers** – Hide or show the Tree and Tune windows. (Hide them for more work area).
13. **Fit Windows To Frame** – Resize all of the object windows to fit into the frame.
14. **Run Analysis** – Run one or more analyses (calculate simulations).
    - When the active document window is a design/schematic and there is only one analysis associated with it, the analysis will be run.
    - If there are several associated analyses then a list containing all the associated analyses (and evaluations) will be displayed, so that the appropriate one may be selected.
    - If no design/schematic is active, or the **drop-down arrow to the right** of the button is clicked, a list containing all the analyses (and evaluations) of the workspace will be displayed, along with options to run all the out-of-date analyses or every analysis in the workspace.
15. **Stop Analyses** – The button is shown instead of the Run Analysis button when any analysis or Evaluation is currently running. Click the button to stop the running Analyses / Evaluations.
    - The drop-down on the right side of the button displays options to Show or Hide the Status Window and to Stop Running Analyses/Evaluations.
16. **Errors Window** – Open the Errors window.

Notes Toolbar

Use this toolbar to edit/modify the Note its text settings. The Notes toolbar automatically displays when you have an active Note.

1. **Font** - select a font for the selection or for typing
2. **Size** - select a font size in html units (3 = average) for the selection or typing
3. **Style** - click the pulldown to pick from standard html styles
4. **Bold** - embolden selected characters
5. **Italic** - italicize selected characters
6. **Underline** - underline selected characters
7. **Color** - select font color
8. **Number** - number the selected paragraphs
9. **Bullet** - bullet the selected paragraphs
10. **Exdent** - exdent a paragraph (reduce indent)
11. **Indent** - indent a paragraph
12. **Left Justify** - left justify the paragraph
13. **Center Justify** - center justify the paragraph
14. **Right Justify** - right justify the paragraph
15. **Image** - Insert a picture into the notes. This picture is specified by a URL.
16. **Absolute** - position part as absolute
17. **Static** - position part as static
18. **Hyperlink** - add a hyperlink (this is currently disabled)

Schematic Toolbar

Use this toolbar to change a schematic or to bring up another toolbar with commonly used parts.

OR

1. **Run** - Runs the analyses.
2. **Part Group** - Show/Hide the part group toolbar.
3. **Annotation** - Show/Hide the Annotation toolbar.
4. **Part Selector** - Show/Hide the Part Selector.
5. **Eye** - Use this pull down menu to turn on/off text displays such as Part Parameters, Net Names, etc. on the schematic.
6. **Keep Connect** - Enable/disable automatic line connections when dragging parts.
7. **Grid Snap** - Enable/disable the grid snap.
8. **Select** - Use the select tool to select parts or annotations.
9. **Pan** - Use the pan tool to pan the schematic around. Press the tool and drag the schematic.
10. **Zoom** - Use the zoom tool to zoom into a rectangular region of the schematic.
11. **Line** - Use the line tool to draw horizontal, vertical or right angled line connections.
12. **Angled Line** - Use the angled line tool to draw line connections of any orientation.
13. **Zoom to Page** - Zoom to fit the page.
14. **Zoom to Fit Selection** - Zoom to fit the currently selected parts/objects on a schematic.
15. **Zoom to Fit All** - Zoom to fit all object.
16. **Tune** - Make the selected parts tunable or fixed.
17. **Disable to short** - Disable/enable the selected parts and simulate them as short circuit.
18. **Disable to open** - Disable/enable the selected parts and simulate them as an open circuit.
19. **Rotate** - Rotate the selected parts by 90 degrees.
20. **Mirror** - Mirror the selected parts.
21. **Open Model or Symbol** - Open part models/symbols. For a single part, this button can open its model/symbol library.

Script Toolbar

Use this toolbar to interact with the active Script and adjust its settings.

1. **Line Numbers** – Hide/Show line numbers on the display.
2. **Script Processor** – Hide/Show the script processor window.
3. **Copy** – copy the script to the script processor.
4. **Run** – copy the script to the script processor and run it.

Spectrasys Toolbar

Use this toolbar to place system parts.

To open: Click the System button on the Schematic Toolbar.

1. **RF Amplifiers (2nd-3rd Order, High Order, Variable Gain)**
2. **Mixers (Basic, Double Balanced, Table)**
3. **Attenuators (Fixed, DC Controlled, Variable)**
4. **Sources (CW, CW with Phase Noise, Wideband, Multicarrier, Intermod, Receiver Intermod, Continuous Frequency, Noise)**
5. **Splitters (2 Way 0 Degree, 2 Way 90 Degree, 2 Way 180 Degree, 3 - 48 Way 0 Degree)**
6. **Switches (SPST, SPDT, SP3T - SP20T)**
7. **Frequency Multipliers (RF Multiplier, RF Divider, Digital Divider)**
8. **Analog to Digital Converter**
9. **Low Pass Filters (Butterworth, Bessel, Chebyshev, Elliptic)**
10. **Band Pass Filters (Butterworth, Bessel, Chebyshev, Elliptic)**
11. **High Pass Filters (Butterworth, Bessel, Chebyshev, Elliptic)**
12. **Band Stop Filters (Butterworth, Bessel, Chebyshev, Elliptic)**
13. **Duplexers (Chebyshev, Elliptic)**
14. **Time Delay**
15. **Phase Shifter**
16. **Ferromagnetic (Circulator, Isolator)**
17. **Couplers (Single Directional, Dual Directional, 90 Degree Hybrid, 180 Degree Hybrid)**
18. **Log Detector**
19. **Oscillator**
20. **Antennas (Coupled, Path)**

Table Toolbar

Use this toolbar to interact with the active Table and adjust its settings.

1. **Properties** - Bring up the properties dialog.
2. **Save** - export/save the table to a file.

See Also

- *Tables* (users)